Broadview
www.broadview.com.cn

[PACKT] open source*
PUBLISHING
community experience distilled

Mastering Microservices with Java

Java微服务

掌握在生产环境下轻松实现微服务的艺术

[美]Sourabh Sharma 著
卢涛 译

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Mastering Microservices with Java

Java微服务

[美]Sourabh Sharma 著
卢涛 译

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

微服务是利用云平台开发企业应用程序的最新技术，它是小型、轻量和过程驱动的组件。微服务适合设计可扩展、易于维护的应用程序。它可以使开发更容易，还能使资源得到最佳利用。本书帮助你用 Java 构建供企业使用的微服务架构，内容包括微服务核心概念和框架、大型软件项目的高层次设计、开发环境设置和前期配置、对微服务架构持续集成的部署、实现微服务的安全性、有效地执行测试、微服务设计的最佳做法和一般原则，以及如何检测和调试问题。

本书适合想要了解微服务架构，以及想要深入了解如何有效地实施企业级微服务的 Java 开发人员。

版权贸易合同登记号　图字：01-2016-7945

图书在版编目（CIP）数据

Java 微服务 /(美) 沙鲁巴 • 夏尔马（Sourabh Sharma）著；卢涛译. —北京：电子工业出版社，2017.1
书名原文：Mastering Microservices with Java

ISBN 978-7-121-30493-4

Ⅰ. ①J… Ⅱ. ①沙… ②卢… Ⅲ. ①JAVA 语言－程序设计 Ⅳ. ①TP312.8

中国版本图书馆 CIP 数据核字（2016）第 288000 号

责任编辑：张春雨
印　　刷：北京中新伟业印刷有限公司
装　　订：北京中新伟业印刷有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　1/16　印张：15.5　字数：322.4 千字
版　　次：2017 年 1 月第 1 版
印　　次：2017 年 1 月第 1 次印刷
定　　价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

本书咨询联系方式：（010）51260888-819，faq@phei.com.cn。

作者简介

Sourabh Sharmahas 具有十年以上的产品/应用程序开发经验。他的专长是开发、部署和测试多层 web 应用程序。他喜欢解决复杂的问题，并寻找最佳的解决方案。

在他的职业生涯中，他已成功地为财富 500 强的客户开发和交付了各种独立应用程序和云应用程序，给他们带来很多收益。

Sourabh 还为他的总部设在美国的顶尖企业产品公司发起并开发了一种基于微服务的产品。他在大学时代，即 20 世纪 90 年代后期，开始编写 Java 程序，而且至今仍然热爱这项工作。

审阅者简介

Guido Grazioli 担任过种类繁多的业务应用程序的开发人员、软件架构师和系统集成人员，他的工作跨越多个领域。他是一位复合型软件工程师，对 Java 平台和工具，以及 Linux 系统管理都有深入了解；对 SOA、EIP、持续集成和交付，以及在云环境中的服务业务流程尤其感兴趣。

目录

前言

微服务（Microservices）架构是软件架构风格的一种。随着云平台的采用，企业应用程序的开发从整体应用程序转移到小型、轻量和过程驱动的组件，这种组件称为微服务。顾名思义，微服务是指小型服务。它们是设计可扩展、易于维护的应用程序的下一个重大事件。它不但使应用程序开发起来更容易，而且还提供了极大的灵活性来以最佳方式利用各种资源。

本书是帮助你构建供企业使用的微服务实现的实践指南。它还解释了领域驱动设计及其在微服务中的采用。它讲述了怎样构建更小型、更轻量、更快速的服务，同时确保其可以很方便地在生产环境中实施。它也讲述了企业应用程序开发从设计与开发，到部署、测试和实现安全性的完整生命周期。

本书包含的内容

第 1 章，一种解决方法，涉及大型软件项目的高层次设计，在生产环境中所面临的共同问题和解决问题的方法。

第 2 章，设置开发环境，讲述了如何设置开发环境，包括 IDE 和其他开发工具，以及不同的库。本章涉及创建基本项目到设置 spring 引导配置，以建立和发展第一个微服务。

第 3 章，领域驱动设计，通过引用一个示例项目为其余的章节设定基调。它使用此示例项目来驱动服务或应用程序的不同功能和领域组合来解释领域驱动设计。

第 4 章，实现微服务，讲述示例项目从设计到实现的过程。本章不仅涉及编码，还涉及微服务的不同方面——构建、单元测试和包装。在本章末尾，将完成一个可用于部署和使用的示例微服务项目。

第 5 章，部署和测试，讲述了如何采用不同的形式，包括独立部署和使用诸如 Docker

的容器来部署微服务。本章还将演示如何用 Docker 把我们的示例项目部署到诸如 AWS 的云服务上面。你还将掌握使用 REST Java 客户端和其他工具来测试微服务的知识。

第 6 章，实现微服务的安全性，解释如何利用身份验证和授权来保证微服务的安全。身份验证将使用基本身份验证和身份验证令牌来讲述。同样，授权将使用 Spring Security 来解释。本章还将解释常见的安全问题及对策。

第 7 章，利用微服务 Web 应用程序来使用服务，解释了如何利用 Knockout、Require 和 Bootstrap JS 库开发 web 应用程序（UI），构建使用微服务来显示数据的 web 应用程序的原型和一个小型实用程序项目（示例项目）的流程。

第 8 章，最佳做法和一般原则，讲述微服务设计的最佳做法和一般原则。本章还提供了有关使用行业做法进行微服务开发的详细信息和范例。本章还包含微服务实现会产生的错误，以及如何才能避免这类问题的几个例子。

第 9 章，故障排除指南，解释了在微服务及其解决方案的开发过程中会遇到的常见问题。这将帮助你顺利地掌握本书内容，并使学习过程轻松。

学习本书需要具备的条件

为了学习本书，可以使用至少具备 2GB 内存的安装了任何操作系统（Linux、Windows 或 Mac）的计算机；还需要 NetBeans with Java、Maven、Spring Boot、Spring Cloud、Eureka Server、Docker 和 CI/CD 的应用程序。对于 Docker 容器，可能需要一个单独的虚拟机或一个云主机，最好拥有 16GB 或更大的内存。

本书的受众

本书面向熟悉微服务架构，并对核心要素和微服务应用程序有一个合理的知识水平和理解，但现在想要深入了解如何有效地实施企业级微服务的 Java 开发人员。

版式约定

你会发现，本书采用了大量的文本样式，用以区分不同种类的信息。下面是这些样式和解释它们的含义的一些例子。

正文中的代码、数据库表名称、文件夹名称、文件名、文件扩展名、路径名、虚拟的URL、用户输入和Twitter句柄如下所示：“可以使用下面的实现创建Table实体，并且可以根据自己的需要添加属性”。

代码块的设置，如下所示：

```
public class Table extends BaseEntity<BigInteger> {

   private int capacity;

   public Table(String name, BigInteger id, int capacity) {
      super(id, name);
      this.capacity = capacity;
   }
```

任何命令行输入或输出采用的格式如下：

```
docker push localhost:5000/sourabhh/restaurant-service:PACKT-SNAPSHOT
docker-compose pull
```

警告或重要的说明在这样的方框中显示。

提示和技巧以这样的形式出现。

下载示例代码

从http://www.broadview.com.cn下载所有已购买的博文视点书籍的示例代码文件。

勘误表

虽然我们已经尽力谨慎地确保内容的准确性，但错误仍然存在。如果你发现了书中的错误，包括正文和代码中的错误，请告诉我们，我们会非常感激。这样，你不仅帮助了其他读者，也帮助我们改进后续的出版。如发现任何勘误，可以在博文视点网站相应图书的

页面提交勘误信息。一旦你找到的错误被证实，你提交的信息就会被接受，我们的网站也会发布这些勘误信息。你可以随时浏览图书页面，查看已发布的勘误信息。

参与本书翻译的人员有卢涛、李颖、卢林、陈克非、李洪秋、张慧珍、李又及、卢晓瑶、李阳、陈克翠、刘雯、汤有四。

1 一种解决方法

作为先决条件，我希望你对微服务和软件架构能有一个基本了解。如果不是这样，我建议你用搜索引擎查一下，在解释并详细介绍它们的众多资源中锁定一个。这将帮助你彻底理解其概念和把握本书内容。

读完这本书，你就可以实现微服务，把它用于本地部署或云生产环境的部署，并学习完整的生命周期，包括设计、开发、测试和部署，以及持续集成和部署。本书是专门为实际使用而编写的，用来激发你作为解决方案架构师的智慧。你的学习将帮助你开发和交付任何类型的本地部署下的产品，包括 SaaS、PaaS，等等。我们将主要使用 Java 和基于 Java 的框架工具，如 Spring Boot 和 Jetty，我们还将使用 Docker 作为容器。

从这里起，本书将用 μServices 来表示 Microservices（微服务），除了用引号括起来的时候。（译者注：实际上只有少数章节用 μServices，因此为了统一，μServices 也译为微服务。）

在本章中，你将学习微服务的永存性及其演化过程。它强调了本地部署和基于云的产品面临的重大问题及微服务如何处理这些问题。本章还解释了在 SaaS、企业级或大型应用程序的开发过程中遇到的常见问题及其解决方案。

在这一章，我们将学习以下主题：

- 微服务和背景简介
- 整体式架构

- 整体式架构的限制
- 微服务提供的灵活性与效益
- 在诸如 Docker 的容器中部署微服务

微服务的演变

Martin Fowler 解释说：

> “‘微服务’一词是 2011 年 5 月在威尼斯附近举办的软件架构师讲习班上讨论的，用来描述参与者所见到的作为一种通用的架构风格的东西，其中有许多已经在最近得到研究。在 2012 年 5 月，同一个小组决定把‘微服务’作为这种东西最适当的名称。”

让我们了解一下它在过去几年的发展的一些背景。企业架构更多地从历史悠久的大型机计算，经过客户端-服务器架构（2 层到 n 层）到**面向服务的架构（service-oriented architecture，SOA）**进化。

从 SOA 到微服务的转变不是一个由任何行业组织定义的标准，而是由许多组织实行的实用方法。SOA 最终进化成为微服务。

Netflix 架构师 Adrian Cockcroft，把它形容为：

> “细粒度 SOA。所以微服务是强调小型短暂组件的 SOA。”

同样，以下引用 Mike Gancarz（X windows 系统的设计团队成员）的叙述，定义了 UNIX 哲学至高无上的一种感受，它同样适合微服务范式：

> “小即是美。”

微服务与 SOA 有许多共同的特点，比如将重点放在服务上，以及如何把一个服务与另一个服务解耦。SOA 通过公开大多基于**简单对象访问协议（SOAP）**的 API，围绕整体应用程序集成展开进化。因此，中间件，比如**企业服务总线（ESB）**，对 SOA 是非常重要的。微服务的复杂度更低，即使它可能使用消息总线，也仅把它用于消息传输并且不包含任何逻辑。

Tony Pujals 漂亮地定义了微服务：

> “在我的心理模型中，我认为微服务是自包含（如在容器中）的轻量进程，它们通过 HTTP 进行通信，用相对较小的工作量与仪式来创建和部署，将只集中在有限领域的 API 提供给它们的使用者。”

整体式架构概述

微服务并不是新事物，它已经流传了很多年。其最近的崛起归功于其声望和知名度的提高。在微服务变得流行之前，被用于开发本地部署和云应用的主要是整体式架构。

整体式架构允许分别开发不同组件，如展示、应用程序逻辑、业务逻辑和**数据访问对象（data access objects，DAO）**，然后你要么将它们一起打包在**企业归档（enterprise archive，EAR）或 web 归档（web archive，WAR）**文件中，要么将它们存储在某个单独的目录层次结构中（例如，Rails、NodeJS，等等）。

许多著名应用程序，如 Netflix，已经使用微服务架构进行开发。此外，eBay、亚马逊和 Groupon 也都已经从整体式架构发展到微服务架构。

现在，你已经深入了解了微服务的背景和历史，接下来让我们讨论一种传统的方法，即整体式应用程序开发的局限性，并将其与微服务解决局限性的方法做比较。

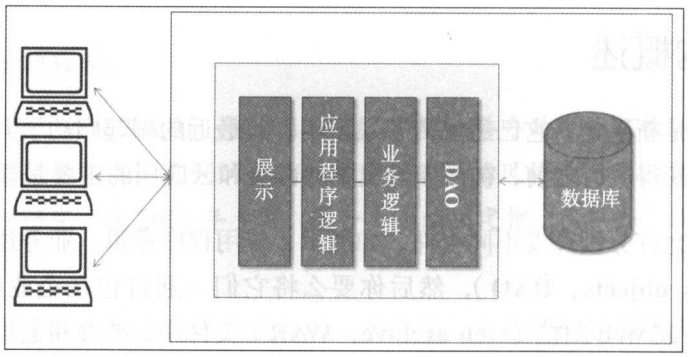
整体式架构的局限性与它的微服务解决方案的对比

正如我们所知，变化是永恒的。人类总是希望寻找更好的解决方案。这就是微服务如何发展成为了它今天的样子的原因，而且它可能会在未来进一步演变。今天，组织使用敏捷方法来开发应用程序，这是一种快节奏的开发环境，在云计算和分布式技术的发明以后，它的规模变得更大。许多人认为整体式架构也可能用于类似用途，并与敏捷方法契合，但微服务还为用于生产的应用程序的许多方面提供了更好的解决方案。

若要了解整体式设计和微服务的差别，让我们举一个餐馆订座应用程序的例子。这个应用程序可能包括很多个服务，如客户、订单、分析等服务，以及常规的组件，如展示和数据库组件。

我们将在这里探索三种不同的设计——传统的整体式设计、使用服务的整体式设计和微服务设计。

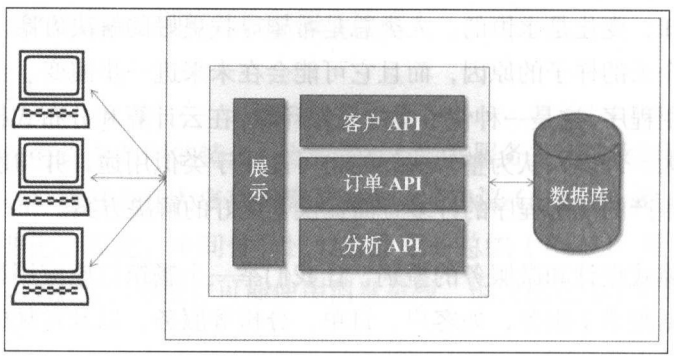
下面的关系图说明了传统的整体式应用程序设计。在 SOA 变得流行之前，这种设计一直被广泛应用：



传统的整体式设计

在传统的整体式设计中，一切都被打包在同一个归档文件中，包括展示代码、应用程序逻辑和业务逻辑代码、DAO 代码、与数据库文件或其他数据源进行交互有关的代码。

在 SOA 产生以后，应用程序开始基于服务来开发，在这种开发模式中，每个组件都为其他组件或外部实体提供服务。下面的关系图描述了提供不同服务的整体式应用程序，在这里，多个服务都被展示组件使用了。所有服务、展示组件或任何其他组件都被打包在一起：

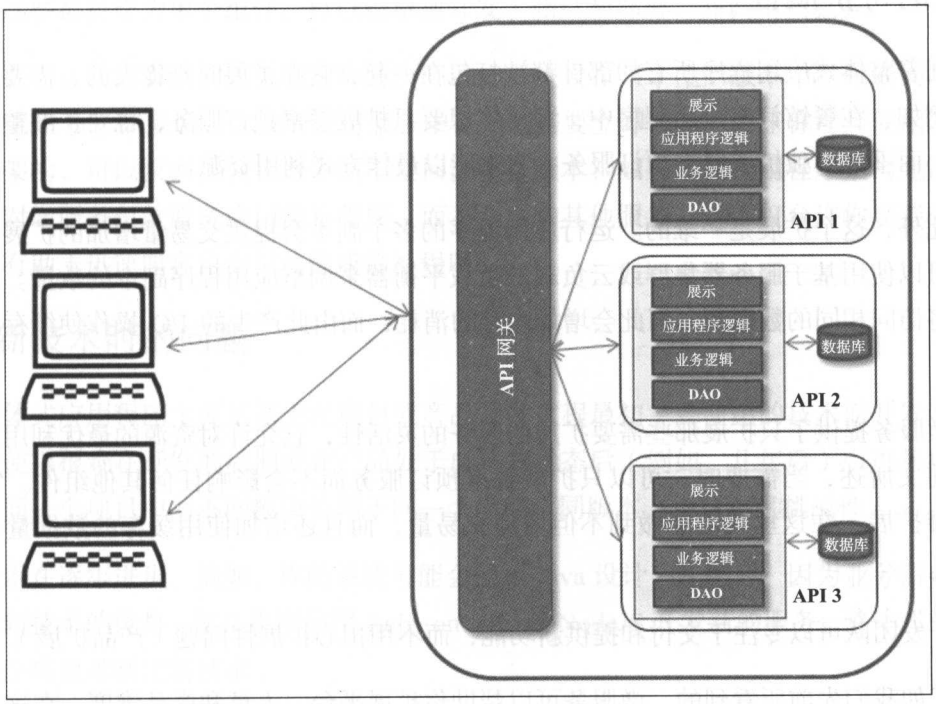


使用服务的整体式设计

下面的第三个设计描绘了微服务。在这里，每个组件都是自主的。每个组件都可以独立开发、构建、测试和部署。在这里，甚至应用程序 UI 组件也可以是一个使用微服务的客户端。在我们的示例中，为了举例说明，在微服务内部使用这个设计的层。

API 网关提供接口，不同的客户端可以通过它访问个别的服务，并解决以下问题：

- 如何让相同的服务给不同的客户端发送不同的响应。例如，预订服务可以发送如下不同的响应、向移动客户端发送最小化的信息、向桌面客户端发送详细信息，它们提供不同的细节，并向第三方客户端发送不同的东西。
- 某个响应可能需要从两个或多个服务中提取信息：



微服务设计

所有的示例设计关系图都是非常高层次的设计，观察了这些之后，你可能会发现，在整体式设计中，各个组件全被捆绑在一起，并且互相紧密耦合。

所有的服务都是相同的捆绑包的一部分。同样，在第二个设计的关系图中，可以看到

第一个关系图的变体，在这个设计中，所有服务都可以有其自己的层，并形成不同的 API，但如图所示，这些也全都捆绑在一起。

相反，在微服务中，设计组件并未捆绑在一起，并且具有松散的耦合。每个服务都有其自己的层，而 DB 被捆绑在一个单独的归档文件中。所有这些已部署的服务都提供它们特定的 API，例如客户、预订或分析。这些 API 都是准备就绪可供使用的，甚至连 UI 也是单独部署，并使用微服务设计的。因此，它比对应的整体式设计具备更多优点。尽管如此，我还是要提醒你，有一些特殊的情况下，整体式的应用程序开发模式是非常成功的，比如 Etsy 和对等电子商务 web 应用程序。

一维的可扩展性

随着整体式应用程序所有的部件都被打包在一起，它在扩展时是庞大的，需要扩展一切。例如，在餐馆订座应用程序中，即使你想要只扩展餐桌预订服务，也要扩展整个应用程序，而不能单独扩展餐桌预订服务。它未能以最佳方式利用资源。

此外，这个扩展是一维的。运行应用程序的多个副本会提供交易量增加的扩展。运营团队可以使用基于服务器集群或云负载的负载平衡器来调整应用程序副本的数量。每个副本都将访问相同的数据源，因此会增加内存的消耗，而由此产生的 I/O 操作使缓存不那么有效。

微服务提供了只扩展那些需要扩展的服务的灵活性，它允许对资源的最优利用。正如我们前文所述，当需要时，可以只扩展餐桌预订服务而不会影响任何其他组件。它还允许二维扩展，在这里我们能做到不但增加交易量，而且还增加使用缓存的数据量（平台扩展）。

开发团队可以专注于交付和提供新功能，而不用担心扩展性问题（产品扩展）。

正如我们先前所看到的，微服务可以帮助你扩展平台、人员和产品维度。在这里的人员扩展指的是，取决于微服务的具体开发和重点需要，扩大或缩小团队规模。

微服务开发使用 REST 式的 web 服务开发，REST 的服务器端是无状态的，使得在这个意义上，它是可扩展的，这意味着服务器之间没有太多的通信，从而它可以水平扩展。

在出故障时回滚版本

因为整体式应用程序是打包在相同的归档文件或包含在单个目录中的，所以这阻碍了代码的模块化部署。例如，很多人可能都曾遇到过由于一项功能的故障，致使整个版本延迟发布的痛苦。

要解决这种问题，微服务给我们提供仅回滚那些出故障的功能的灵活性。这是一种非常灵活和富有成效的方法。例如，假设你是在线购物门户开发团队的成员，想要开发一个基于微服务的应用程序。可以将你的应用程序基于不同的领域，如商品、付款、购物车等进行划分，并将所有这些组件都作为单独的程序包打包。一旦你已经分别部署所有这些软件包，这些都将作为单个组件，可以被单独开发、测试和部署，并调用微服务。

现在，让我们看看它是如何帮你解决问题的。假设在某个生产版本中发布新的功能、增强功能和 bug 修复后，你发现支付服务中有缺陷需要立即修复。由于你已经使用基于微服务的架构，可以仅回滚付款服务而不是回滚整个版本，如果你的应用程序架构允许，还可以只对微服务付款服务应用修补程序，而不会影响其他服务。这不仅允许你妥善处理故障，还有助于迅速向客户交付功能或修补程序。

采用新技术时的问题

整体式应用程序主要是基于在项目或产品开发过程最初主要使用的技术而开发和加强的。这使得很难在开发的后期或在产品处于成熟的状态后（例如，几年后）引进新技术。此外，同一个项目中的不同模块依赖于同一个库的不同版本，使这更具挑战性。

技术在逐年进步。例如，你的系统可能会使用 Java 设计，几年后，因为业务需要或为了利用新技术的优势，你又想要使用 Ruby on rails 或 NodeJS 开发新的服务。整体式的现有应用程序将很难利用新技术。

这不只是代码级集成，还与测试和部署相关。虽然可以通过重新编写整个应用程序来采用一项新技术，但这要耗费大量时间并且具有很高的风险。

另一方面，由于微服务是基于组件开发和设计的，它给我们提供了在开发中使用无论是新技术还是旧技术的任何技术的灵活性。它并不限制你使用特定的技术，这给你提供

了一种开发和工程活动的新范式。可以在任何时候使用 Ruby on Rails、NodeJS 或任何其他技术。

那么，它是如何实现的呢？嗯，非常简单。基于微服务的应用程序代码并不打包成一个单一的归档文件，也并不存储在单个目录中。每个微服务都有其自己的文件，并且单独部署。一个新服务可以在隔离的环境中开发，并可以没有任何技术问题地被测试和部署。正如你所知，微服务还拥有自己独立的进程，它服务于它的目标时不存在任何冲突，如共享紧密耦合的资源，并且进程也保持独立。

因为根据定义，微服务是小型、自包含的功能，所以它提供了低风险地尝试一项新技术的机会。而对整体式系统而言，绝对不能这样。

还可以把你的微服务作为开放源码软件对外提供，所以它可由其他人使用。因此，如果需要，它能够与封闭源码的专有服务进行交互，而使用整体式应用程序，是不可能这么做的。

与敏捷实践的契合

使用敏捷实践开发整体式应用程序是没有问题的，而且这些应用程序正在被开发中。**持续集成（Continuous Integration，CI）**和**持续部署（Continuous Deployment，CD）**也可以使用，但是，问题是——它能有效地使用敏捷实践吗？让我们来研究以下几点：

- 例如，当事件互相依赖的可能性很大，并可能有不同的场景时，一个事件只有在它所依赖的事件都已完成后才能发生。
- 随着代码量的增加，生成应用程序需要更多的时间。
- 大型整体式应用程序要实现频繁部署是一项困难的任务。
- 即使更新单个组件，也将不得不重新部署整个应用程序。
- 重新部署可能导致已经运行的组件出问题，例如作业调度器可能会更改，无论组件是否对它产生影响。
- 如果单个更改的组件不能正常工作，或如果它需要更多的修正，重新部署的风险可能会增加。
- UI 开发人员总是需要更多的重新部署，这对于整体式的大型应用程序是相当危险和费时的。

前面的问题都可以通过微服务很容易地解决。例如，UI 开发人员可能拥有他们自己的 UI 组件，这些组件能够单独地开发、构建、测试和部署。同样，其他的微服务也可能独立部署，并且因其自主的特点，降低了系统故障风险。它用于开发的另一个优势是，UI 开发人员可以使用 JSON 对象和模拟 Ajax 调用来开发 UI，这些能够以隔离的方式占用。开发完成后，开发人员可以使用实际的 API 并测试这些功能。总之，可以说，微服务开发是快捷的，它符合企业的增量需求。

减轻开发工作量——可以做得更好

一般来说，大型整体式应用程序的代码是开发人员最难理解的，并且新开发人员需要相当长的时间，才能成为生产力。甚至把大型整体式应用程序加载到 IDE 中都是麻烦的，它会降低 IDE 的运行速度，使开发人员的生产力降低。

大型整体式应用程序中的更改很难实施，并花费更多的时间。这是由于有大量的代码库，并且，如果未恰当并彻底地执行影响分析，存在 bug 的风险会很高。因此，执行全面影响分析成为开发人员实施更改之前的先决条件。

在整体式应用中，随着时间的推移，当所有组件都捆绑在一起时，依赖关系就建立了。因此，随着代码更改量（修改后的代码的行数）的增加，与代码更改相关联的风险呈指数级上升。

当一个代码库非常庞大并且有 100 多个开发人员正在它上面工作时，因为前面提到的原因，建立产品和实现新功能变得很困难。你需要确保一切都到位，并且一切都协调一致。在这种情况下，精心设计和记录的 API 的帮助很大。

Netflix，一家互联网流媒体按需提供商，在让约 100 人开发他们的应用程序时遇到了问题。然后，他们使用云平台，并将其应用程序分解成单独的小块。这些最终成为了微服务。提高速度和灵活性以便部署团队独立的愿望导致微服务不断壮大。

微型组件被制作成松耦合的，这归功于 API 的公开，这些 API 可以被持续地集成和测试。借助微服务的连续发布周期，变化被控制得很小，开发人员可以迅速利用它们执行回归测试，然后复审一遍并修复最终发现的瑕疵，以减少部署的风险。这会获得更快的速度与较低的相关风险。

由于功能分离和单一责任原则，微服务使团队非常有成效。可以在网上找到大量的大型项目由人数很少，如八至十个开发人员的团队开发完成的实例。

开发人员对于更少量的代码可以更集中注意力，由此产生的功能的更好实现，导致与产品的用户有更高的移情关系。这有助于功能的实现有更好的动机和明确性。与用户的移情关系会产生一个较短的反馈回路，以及更好、更迅速确定优先次序的功能管道。较短反馈回路使得缺陷检测也更快。

每个微服务团队都独立地工作，而无须与更多的观众协调就可以实现新功能或想法。在微服务设计中，端点故障处理也很容易实现。

最近，在一次会议中，一个团队展示了他们是如何在 10 周内开发出一个具有超级类型（Uber-type）跟踪功能，并包括 iOS 和 Android 应用程序基于微服务的传输-跟踪的应用程序。一家大型咨询公司针对相同的应用程序为他的客户给出 7 个月的估计开发时间。它表明了微服务与敏捷方法和 CI/CD 的契合方式。

微服务的构建管道

微服务也可以使用诸如 Jenkins、TeamCity 等流行的 CI/CD 工具建立和测试。它与整体式的应用程序的构建方式是非常相似的。在多个微服务中，每个微服务都被当作一个小型应用程序。

例如，一旦把代码提交到存储库（SCM）中，CI/CD 工具就触发构建过程：

- 清理代码
- 代码编译
- 执行单元测试
- 建立应用程序归档文件
- 在开发、QA 等各种服务器上部署
- 执行功能和集成测试
- 创建映像容器
- 任何其他步骤

然后，更改 `pom.xml`（在 Maven 的情况下）中的快照（SNAPSHOT）或发行

（RELEASE）版本的版本构建触发器，会构建工件，如同在正常构建触发器中所述的那样。将工件发布到 artifacts 存储库。在存储库中标记此版本。如果你使用容器映像，那么就会生成容器映像。

使用诸如 Docker 的容器部署

由于微服务的设计，需要有一个提供灵活性、敏捷性和平滑度的环境，以支持持续集成和部署，以及发布。微服务部署需要速度、隔离管理和敏捷的生命周期。

产品和软件也可以使用多式联运集装箱（intermodal-container）模型的概念来发布。多式联运集装箱是一个大型的标准化容器，用来执行多式联运货物运输。它允许货物使用各种交通工具——卡车、铁路或轮船运输，而不需要卸载并重新加载。这是储存和运输货物有效和安全的方式。它解决了航运的一个难题，这以前一直是一个耗时的劳动密集型过程，而重复的处理往往会打破易碎物品。

航运集装箱封装它们的内容。同样，软件容器也开始用于封装它们的内容（产品、应用程序、依赖关系，等等）。

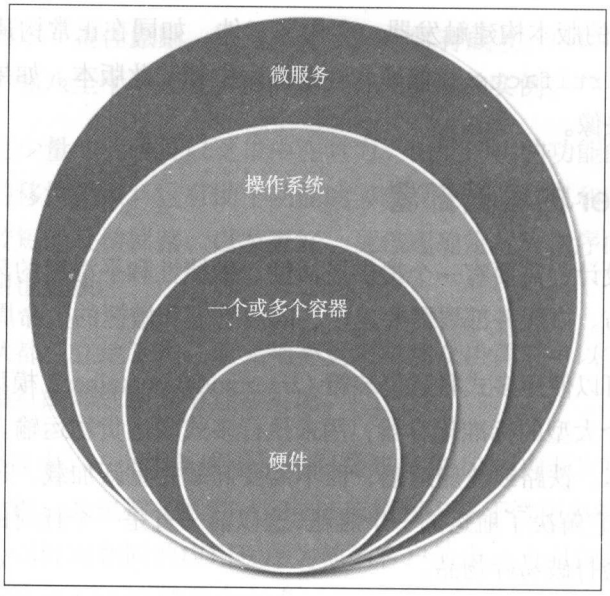
以前，**虚拟机（virtual machine，VM）**被用于创建可以在需要的地方部署的软件映像。后来，诸如 Docker 的容器变得更受欢迎，因为它们同时与传统虚拟站系统和云环境兼容。例如，在开发人员的笔记本电脑上部署超过两个 VM 不太现实。建立和启动一个 VM 机器通常是 I/O 密集型的操作，因此也相当缓慢。

容器

容器（例如，Linux 容器）提供轻量级运行时环境，该环境由虚拟机的核心功能和操作系统的隔离服务组成。这使得包装和执行微服务变得简单和顺利。

如下图所示，容器作为应用程序（微服务）在操作系统中运行。操作系统位于硬件的顶部，每个操作系统都可以有多个容器，使用一个容器运行应用程序。

容器能利用操作系统的内核接口，如 cname 和命名空间，它们允许共享相同内核的多个容器可以完全互相隔离地同时运行。这具有无须为每个用途而完成一个 OS 安装的好处，从而可以消除开销。这也使得硬件得到最佳的利用。



容器的层次关系图

Docker

容器技术是当今发展最迅速的技术之一，而 Docker 引领这一技术。Docker 是一个开放源码项目，2013 年发布。2013 年 8 月其交互式教程推出后，有 1 万个开发人员尝试使用它。1.0 版本在 2013 年 6 月发布的时候，被下载了 275 万次。许多大公司，如微软、RedHat、惠普、OpenStack，以及诸如亚马逊网络服务、IBM 和谷歌等服务提供商，都已与 Docker 签署了伙伴关系协定。

正如我们刚才提到的，Docker 也利用了 Linux 内核的功能，例如 cgroups 和命名空间，以确保资源隔离，并把应用程序及其依赖项打包在一起。这种依赖项的包装使应用程序能够跨不同 Linux 操作系统/发行版本正常运行，从而实现可移植性级别的支持。此外，这种可移植性允许开发人员用任何一种语言开发一个应用程序，然后轻松地把它从笔记本电脑部署到测试或生产服务器上。

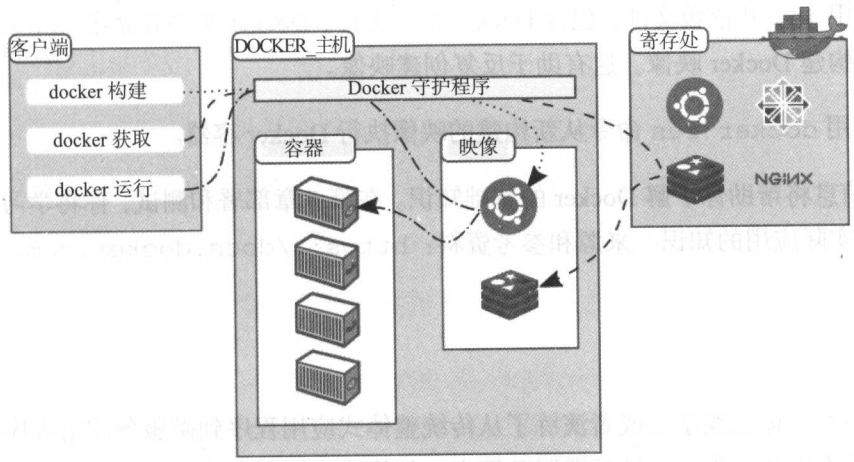
Docker 可原生地在 Linux 上运行。然而，利用 VirtualBox 和 boot2docker，Docker 也可以运行在 Windows 和 Mac OS 上。

容器仅由应用程序及其依赖项组成，依赖项包括基本的操作系统。这使得它轻量，并在资源利用率方面保持高效。开发人员和系统管理员对容器的可移植性和资源的有效利用感兴趣。

Docker 容器中的所有程序都可原生地在主机上执行，并直接使用主机内核。每个容器都具有其自己的用户命名空间。

Docker 的架构

如 Docker 文档指出的，Docker 架构使用客户机-服务器架构。如下面的图例所示（源自 Docker 的网站），Docker 客户端主要是由最终用户使用的用户界面，客户端与 Docker 守护进程之间往复通信。Docker 守护程序承担构建、运行和分发你的 Docker 容器的繁重任务。Docker 客户端和守护进程既可以在同一个系统中，也可以在不同的机器上运行。Docker 客户端与守护进程通过套接字或通过基于 REST 的 API 进行通信。Docker 寄存处是公共或私有 Docker 映像存储库，你可以从中上传或下载映像，例如 Docker 枢纽（`hub.docker.com`）就是一个公共的 Docker 寄存处。



Docker的架构

Docker 的主要组件是一个 Docker 映像和一个 Docker 容器。

Docker 映像

Docker 映像是一个只读的模板。例如，映像可以包含安装了 Apache web 服务器和 web 应用程序的 Ubuntu 操作系统。Docker 映像是 Docker 生成组件。映像用于创建 Docker 容器。Docker 提供简单的方式来生成新的映像或更新现有的映像。你还可以使用由其他人创建的映像。

Docker 容器

Docker 容器是从某个 Docker 映像创建的。Docker 的工作是使得容器只能看到它自己的进程，并有分层到主机文件系统和网络栈的自己的文件系统，它通过管道连接到主机网络栈。Docker 容器可以被运行、启动、停止、移动或删除。

部署

使用 Docker 部署的微服务处理三个部分的问题：

1. 应用程序打包，例如 `jar`
2. 使用 `jar` 和依赖文件，包含 Docker 指令文件、Docker 文件和命令 `docker build` 来构建 Docker 映像。这有助于反复创建映像。
3. 使用 `docker run` 命令从新构建的映像执行 Docker 容器。

上述信息将帮助你了解 Docker 的基础知识。在第 5 章部署和测试，你将学习到更多关于 Docker 实际应用的知识。来源和参考资料：`https://docs.docker.com`。

小结

在这一章，你已经学会或者演练了从传统整体式应用程序到微服务应用程序的大型软件项目高层次设计。你也了解了微服务简史、整体式应用程序的限制和好处，及微服务提供的灵活性。我希望这一章能帮助你理解整体式应用程序在生产环境中面临的共同问题，而微服务可以解决这些问题。你还了解了轻量级和高效的 Docker 容器，并看到容器化为何是一种很好的部署微服务的典型方法。

在下一章中，你会学习使用 IDE 和其他开发工具，针对不同的库来设置开发环境的相关知识。我们将面对创建基本项目和设置 Spring Boot 配置，建立和开发我们第一个微服务。在这里，我们将采用 Java 8 作为开发语言，并把 Spring Boot 用于我们的项目。

2

设置开发环境

本章着重介绍开发环境的设置和配置。如果你熟悉这些工具和库，那么你可以跳过这一章，继续阅读**第 3 章**，在第 3 章你可以研究领域驱动设计的课题。

这一章将涵盖以下主题：

- Spring Boot 配置
- 示例 REST 程序
- 生成安装程序
- 使用 Postman Chrome 扩展执行 REST API 测试
- NetBeans——安装和设置

本书将仅使用开放源代码工具和框架来举例和编码，也将使用 Java 8 作为其编程语言，而应用程序框架将基于 Spring 框架。本书使用 Spring Boot 来开发微服务。

NetBeans 提供最先进的、同时支持 Java 和 JavaScript 的**集成开发环境（NetBeans Integrated Development Environment，IDE）**，足以满足我们的需求。它多年来演变了很多，并已内置支持大多数本书使用的技术，如 Maven、Spring Boot 等。因此，建议你使用 NetBeans IDE。然而，你可以随意使用任何 IDE。

我们将使用 Spring Boot 来开发 REST 服务和微服务。在本书中选择最流行的 Spring 框架——Spring Boot 或其子集 Spring Cloud 是有意为之。因为这样我们就不需要从头开始编

写应用程序，该框架对云应用程序大部分的东西都提供默认的配置。在 Spring Boot 配置一节中，我们对 Spring Boot 做了概述。如果你是 Spring Boot 初学者，那么这会对你有帮助。

我们将使用 Maven 作为我们的构建工具。与 IDE 的情况一样，可以随意使用任何构建工具，例如 Gradle 或 Ant，我们将使用嵌入式的 Jetty 作为 web 服务器，但另一种选择是使用嵌入式的 Tomcat web 服务器。我们还将使用 Chrome 的 Postman 扩展来测试 REST 服务。

我们将从 Spring Boot 配置开始介绍。如果你是 NetBeans 初学者或者正面临设置环境的问题，可以参考最后一节 NetBeans IDE 安装部分的解释，否则完全可以跳过该节。

Spring Boot 配置

为了开发最先进且可用于生产的特定于 Spring 的应用程序，Spring Boot 是一个明显的选择。其网站还说明了其真正的优势：

> “需要坚持构建可用于生产的 Spring 应用程序的观念。Spring Boot 约定优先于配置，并且让你尽可能快地启动并运行。”

Spring Boot 概述

Spring Boot 是 Pivotal 创建的优异的 Spring 工具，它于 2014 年 4 月发布（GA）。它基于 SPR-9888（`https://jira.spring.io/browse/SPR-9888`）的请求被开发出来，该请求标题为“对无容器 web 应用程序架构的改进支持”。

你一定会想，为什么是无容器呢？因为，今天的云环境或 PaaS 都提供基于容器 web 架构的大多数功能，如可靠性、可管理性或可扩展性。因此，Spring Boot 侧重于使其本身成为超轻量的容器。

Spring Boot 预先配置使得开发可用于生产的 web 应用程序变得非常容易。Spring Initializer（初始值设定项）（`http://start.spring.io`）是一个网页，可以在其中选择构建工具如 Maven 或 Gradle，还可以选择项目元数据，如组、工件和依赖项。当填充了所需的字段后，只需单击 **Generate Project（生成项目）**按钮，它就会提供一个可以用于生产

应用程序的 Spring Boot 项目。

在此页上，默认打包选项是 jar。我们也会在微服务开发中使用 jar 包装。原因非常简单：它使得微服务的开发更容易。只要想想，管理和创建每个微服务都运行在其自己的服务器实例上的基础设施会多么困难，就明白了。

Josh Long 在某个 Spring IOs 的谈话中分享了他的观点：

> “最好生成 Jar，而不是 War。”

稍后，我们将使用 Spring Cloud，它是在 Spring Boot 之上的一个包装。

把 Spring Boot 添加至 REST 示例

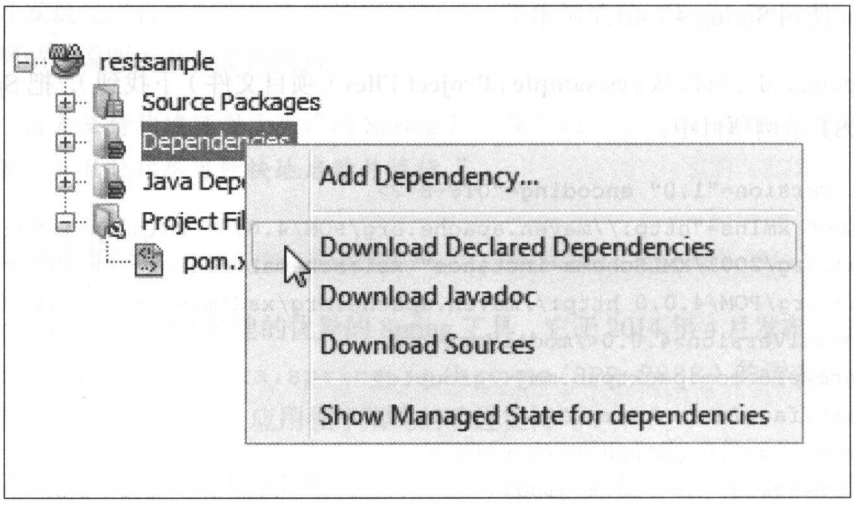
在编写本书的时候，Spring Boot 提供了 1.2.5 发布版本，你可以使用最新的发布版本。Spring Boot 使用 Spring 4（4.1.7 版本）。

打开 pom.xml（可以从 **restsample | Project Files**（项目文件）下找到），把 Spring Boot 添加到 REST 示例项目中：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.
apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <groupId>com.packtpub.mmj</groupId>
   <artifactId>restsample</artifactId>
   <version>1.0-SNAPSHOT</version>
   <packaging>jar</packaging>
   <parent>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-parent</artifactId>
      <version>1.2.5.RELEASE</version>
   </parent>
   <properties>
```

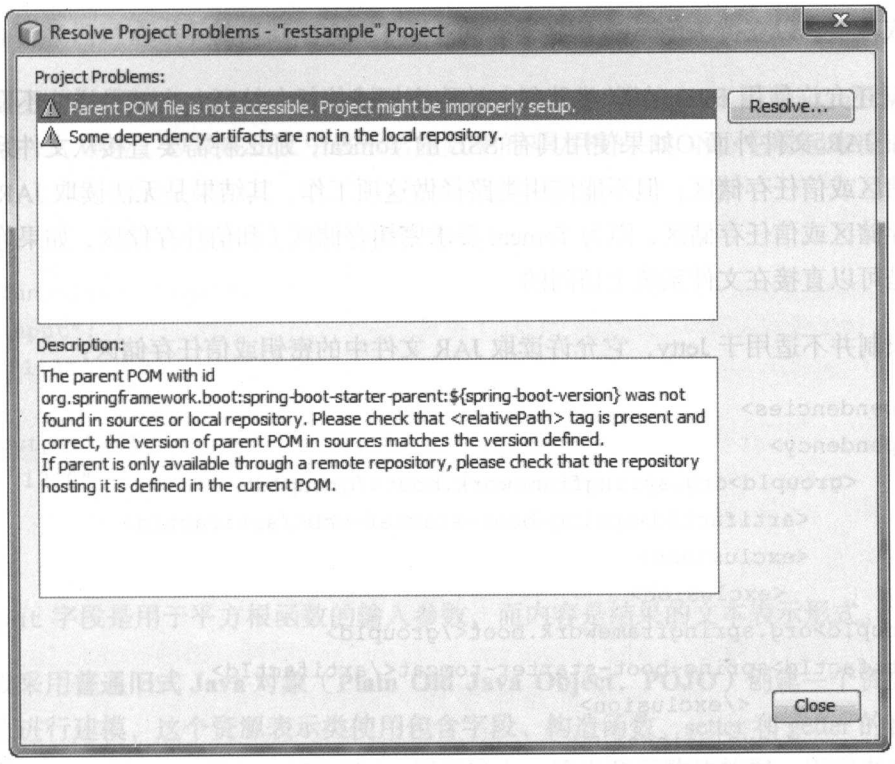
```
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <spring-boot-version>1.2.5.RELEASE</spring-boot-version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
            <version>${spring-boot-version}</version>
        </dependency>
    </dependencies>
</project>
```

如果你是第一次添加这些依赖项，需要在如下所示的 **Projects** 窗格的 **restsample** 项目中通过鼠标右键单击 Dependencies 文件夹来下载依赖项。



下载NetBeans的Maven依赖项

同样，要解决项目问题，用鼠标右键在 NetBeans 项目 **restsample** 上单击，并选择 **Resolve Project Problems…**（**解决项目问题**），它将打开如下图所示的对话框。单击 **Resolve…**按钮来解决问题。



解决项目问题对话框

如果在代理服务器后面使用 Maven，那么需要更新<NetBeans 安装目录>\java\maven\conf\settings.xml 中的代理服务器设置。可能需要重新启动 NetBeans IDE 才能使更新的设置生效。

如果在本地 Maven 存储库中未提供声明的依赖项和传递依赖项，那么前面的步骤将从远程 Maven 资源库中下载所有必需的依赖项。如果是第一次下载依赖项，那么可能需要花一些时间，这具体取决于你的 Internet 速度。

添加一个嵌入式 Jetty 服务器

Spring Boot 默认情况下提供 Apache Tomcat 作为嵌入式应用程序的容器。本书将在使用 Apache Tomcat 的地方使用嵌入式 Jetty 应用程序容器。因此，我们需要添加 Jetty 应用程

序容器依赖项，以支持 Jetty web 服务器。

Jetty 还允许使用 classpath（类路径）读取密钥或信任存储区，也就是说，不需要把这些存储在 JAR 文件外面。如果使用具有 SSL 的 Tomcat，那么将需要直接从文件系统访问密钥存储区或信任存储区，但不能使用类路径做这项工作。其结果是无法读取 JAR 文件中的密钥存储区或信任存储区，因为 Tomcat 要求密钥存储区（和信任存储区，如果你正在使用它）是可以直接在文件系统上访问的。

此限制并不适用于 Jetty，它允许读取 JAR 文件中的密钥或信任存储区：

```
<dependencies>
<dependency>
      <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-web</artifactId>
          <exclusions>
            <exclusion>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-tomcat</artifactId>
              </exclusion>
           </exclusions>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
</dependencies>
```

示例 REST 程序

我们将使用一个简单的方法来建立一个独立的应用程序。我们把这一切都打包成一个由 `main()` 方法驱动的单独的可执行 JAR 文件。在此过程中，使用 Spring 对嵌入 Jetty servlet 容器的支持作为 HTTP 运行时环境，而不是将它部署到一个外部的实例中。因此，我们将创建可执行的 JAR 文件来代替需要在外部 web 服务器上部署的 WAR 文件。

现在，当你在 NetBeans IDE 中准备好 Spring Boot 后，就可以创建你的示例 web 服务了。你将创建一个数学 API，它可以执行简单的计算并生成 JSON 格式的结果。

下面让我们讨论一下如何调用 REST 服务并获取其响应。

此服务将处理对/calculation/sqrt 或/calculation/power 等的 GET 请求。GET 请求应该返回 200 OK 响应，以及在正文中表示给定数字的平方根的 JSON。它看起来应该像下面这样：

```
{
  "function": "sqrt",
  "input": [
    "144"
  ],
  "output": [
    "12.0"
  ]
}
```

input 字段是用于平方根函数的输入参数，而内容是结果的文本表示形式。

可以采用**普通旧式 Java 对象（Plain Old Java Object，POJO）**创建一个资源表示类对表示法进行建模，这个资源表示类使用包含字段、构造函数、setter 和 getter 的普通旧式 Java 对象 Plain Old Java Object（POJO），用于输入、输出和函数的数据。用它来对表示法进行建模：

```
package com.packtpub.mmj.restsample.model;

import java.util.List;

public class Calculation {

    String function;
    private List<String> input;
    private List<String> output;

    public Calculation(List<String> input, List<String> output, String
function) {
        this.function = function;
        this.input = input;
```

```
        this.output = output;
    }
    public List<String> getInput() {
        return input;
    }

    public void setInput(List<String> input) {
        this.input = input;
    }

    public List<String> getOutput() {
        return output;
    }

    public void setOutput(List<String> output) {
        this.output = output;
    }

    public String getFunction() {
        return function;
    }

    public void setFunction(String function) {
        this.function = function;
    }

}
```

编写 REST 控制器类

Roy Fielding 在其博士论文中提出并定义了术语 **REST（Representational State Transfer，具象状态转换）**。REST 是一种用于诸如 WWW 的分布式超媒体系统的软件架构风格，RESTful（REST 式）是指那些符合 REST 架构属性、原则和约束的系统。

现在，你将创建 REST 控制器来处理计算资源。在 Spring REST 式的 web 服务实现中，控制器负责处理 HTTP 请求。

@RestController

@RestController 是 Spring 4 中引入的用于 resource 类的类级注解。它是@Controller 与@ResponseBody 的组合，因此类返回一个域对象，而不是视图。

在以下代码中，可以看到 CalculationController 类通过返回 calculation 类的一个新实例来处理对/calculation 的 GET 请求。

我们将实现两个计算资源的 URL——将平方根（Math.sqrt()）函数实现为/calculation/sqrt URL，以及将幂（Math.pow()）函数实现为/calculation/power URL。

@RequestMapping

@RequestMapping 注解在类级别使用，它将/calculation URI 映射到 CalculationController 类，它确保对/calculation 的 HTTP 请求被映射到 CalculationController 类。基于使用 URI 的注解@RequestMapping 定义的路径（/calculation 后缀，例如/calculation/sqrt/144），会映射到各自的方法。在这里，映射到/calculation/sqrt 的请求被映射到 sqrt()方法，而映射到/calculation/power 的请求被映射到 pow()方法。

你可能也已观察到我们并没有定义这些方法会使用什么请求方法（GET/POST/PUT 等等）。@RequestMapping 注解默认映射所有的 HTTP 请求方法。可以通过使用 RequestMapping 的方法属性来使用特定的方法。例如，可以通过以下方式使用 POST 方法来编写@RequestMethod 注解:

```
@RequestMapping(value = "/power", method = POST)
```

为传递过程中的参数，此示例说明请求参数和路径参数都分别使用@RequestParamand 和@PathVariable 注解。

@RequestParam

@RequestParam 负责将查询参数绑定到控制器方法的参数。例如，QueryParam 底数和指数分别绑定到 CalculationController 的 pow()方法的参数 b 和参数 e 上。pow()方法的两个查询参数都是必需的，因为我们未对它们使用任何默认值。可以使用

@RequestParam 的 defaultValue 属性来设置查询参数的默认值，例如@RequestParam (value="base",defaultValue="2")，在这里，如果用户未传递查询参数底数，那么底数将使用默认值 2。

如果没有定义 defaultValue，并且用户未提供请求的参数，则 RestController 返回 HTTP 状态代码 400，以及 **400 所需的字符串参数底数不存在（Required String parameter base is not present）**的消息。如果缺少多个请求参数中的一个，它总是使用所需的第一个参数的引用：

```
{
  "timestamp": 1464678493402,
  "status": 400,
  "error": "Bad Request",
  "exception": "org.springframework.web.bind.
MissingServletRequestParameterException",
  "message": "Required String parameter 'base' is not present",
  "path": "/calculation/power/"
}
```

@PathVariable

@PathVariable 可以帮助你创建动态的 URI。@PathVariable 注解允许你将 Java 参数映射到一个路径参数。它与@RequestMapping 配合工作，其中后者在 URI 中创建占位符，然后要么作为 PathVariable，要么作为方法的参数使用相同的占位符名称，正如可以在 CalculationController 类方法 sqrt()中看到的。在这里，值占位符是在@RequestMapping 内创建的，并且相同的值被赋予@PathVariable 的值。

sqrt()方法提取 URI 中的参数来代替请求的参数。例如，http://localhost:8080/calculation/sqrt/144。在这里，值 144 作为路径参数传递，而此 URL 应该返回 144 的算术平方根，也就是 12。

为了使用现成的基本检查，我们使用正则表达式"^-?+\\d+\\.?+\\d*$"来只允许有效的数字作为参数。如果传递了非数值，那么每个方法都在 JSON 的输出键中添加一条错误消息。

CalculationController 也使用正则表达式，在 path 变量（path 参数）中的.+允许/path/{variable:.+}的数值中带小数点(.)。Spring 将忽略最后一个点号后面的任何东西。Spring 的默认行为把它当作文件扩展名。

还有其他替代办法，如在末尾添加一个正斜杠(/path/{variable}/)或通过把 useRegisteredSuffixPatternMatch 设置为 true，使用 PathMatchConfigurer(在 Spring 4.0.1 及更高版本中可用)重写 WebMvcConfigurerAdapter 的 configurePathMatch() 方法。

```
package com.packtpub.mmj.restsample.resources;
```

```
package com.packtpub.mmj.restsample.resources;

import com.packtpub.mmj.restsample.model.Calculation;
import java.util.ArrayList;
import java.util.List;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import static org.springframework.web.bind.annotation.RequestMethod.GET;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/calculation")
public class CalculationController {

    private static final String PATTERN = "^-?+\\d+\\.?+\\d*$";

    @RequestMapping("/power")
    public Calculation pow(@RequestParam(value = "base") String b, @
RequestParam(value = "exponent") String e) {
        List<String> input = new ArrayList();
        input.add(b);
        input.add(e);
```

```
        List<String> output = new ArrayList();
        String powValue = "";
        if (b != null && e != null && b.matches(PATTERN) && e.matches(PATTERN)) {
            powValue = String.valueOf(Math.pow(Double.valueOf(b), Double.
valueOf(e)));
        } else {
            powValue = "Base or/and Exponent is/are not set to numeric value.";
        }
        output.add(powValue);
        return new Calculation(input, output, "power");
    }

    @RequestMapping(value = "/sqrt/{value:.+}", method = GET)
    public Calculation sqrt(@PathVariable(value = "value") String aValue)
{
        List<String> input = new ArrayList();
        input.add(aValue);
        List<String> output = new ArrayList();
        String sqrtValue = "";
        if (aValue != null && aValue.matches(PATTERN)) {
            sqrtValue = String.valueOf(Math.sqrt(Double.valueOf(aValue)));
        } else {
            sqrtValue = "Input value is not set to numeric value.";
        }
        output.add(sqrtValue);
        return new Calculation(input, output, "sqrt");
    }
}
```

在这里，我们只使用 URI/calculation/power 和/calculation/sqrt 来公开 Calculation 资源的 power 和 sqrt 函数。

在这里，我们使用 sqrt 和 power 作为我们 URI 的一部分，仅用于演示目的。理想情况下，这些应该已经被作为请求参数"function"的值来传递，或基于端点设计形成的类似东西。

一个有趣的事情是，由于 Spring 的 HTTP 消息转换器的支持，Calculation 对象被自动转换为 JSON，不需要手动执行此转换。如果 Jackson 2 位于类路径中，那么 Spring 的 MappingJackson2HttpMessageConverter 就会把 Calculation 对象转换为 JSON。

制作一个示例 REST 可执行应用程序

创建一个类 RestSampleApp 和 SpringBootApplication 注解。main()方法使用 Spring Boot 的 SpringApplication.run()方法来启动应用程序。

我们将使用@SpringBootApplicationthat 来注解 RestSampleApp 类，它隐式添加所有下列标记：

- @Configuratio 注解把这个类标记为应用程序上下文定义的 Bean 源。
- @EnableAutoConfiguration 注解指示，Spring Boot 开始基于 classpath 设置，其他 Bean 类和各种属性设置来添加 bean。
- 如果 Spring Boot 在 classpath 上发现 springwebmvc，@EnableWebMvc 注解就被添加。它将此应用程序视为一个 web 应用程序，并激活设置 DispatcherServlet 等关键行为。
- @ComponentScan 注解告诉 Spring 在给定包中寻找其他的组件、配置和服务：

```
package com.packtpub.mmj.restsample;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.
SpringBootApplication;

@SpringBootApplication
public class RestSampleApp {

    public static void main(String[] args) {
        SpringApplication.run(RestSampleApp.class, args);
    }
}
```

这个 web 应用是 100%纯 Java 的，你不必使用 XML 来处理任何管道或基础设施的配置，相反，它使用 Java 注解，Spring Boot 甚至使之变得更简单。因此，除了用于 Maven 的 pom.xml 外，没有一行 XML，甚至没有 web.xml 文件。

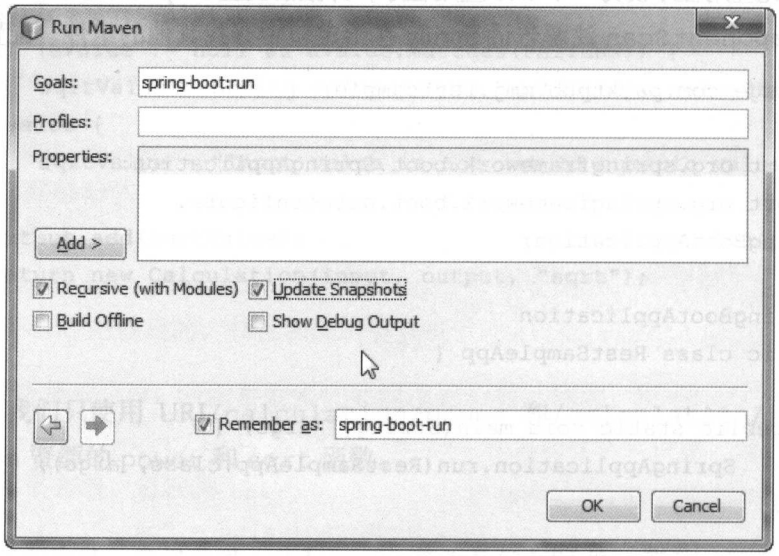
设置应用程序构建

直到现在，我们使用的 pom.xml 足以执行示例 REST 服务。这个服务的代码将打包进一个 JAR 文件里。要使这个 JAR 成为可执行文件，我们需要选择以下选项。

运行 Maven 工具

在这里，我们使用 Maven 工具来执行生成的 JAR，步骤如下：

1. 用鼠标右键单击 pom.xml。
2. 从弹出的快捷菜单中选择 **run-maven | Goals…**，它将打开对话框。在 **Goals** 字段中键入 spring-boot:run。我们已经在代码中使用了 Spring Boot 的正式发行版本。然而，如果你正在使用快照版本，可以选中 **Update Snapshots** 复选框。若要在将来使用它，请在 **Remember as** 字段中键入 spring-boot-run。
3. 下一次，可以直接单击 **run-maven | Goals | spring-boot-run** 执行该项目。



运行Maven对话框

4. 单击 **OK** 按钮执行该项目。

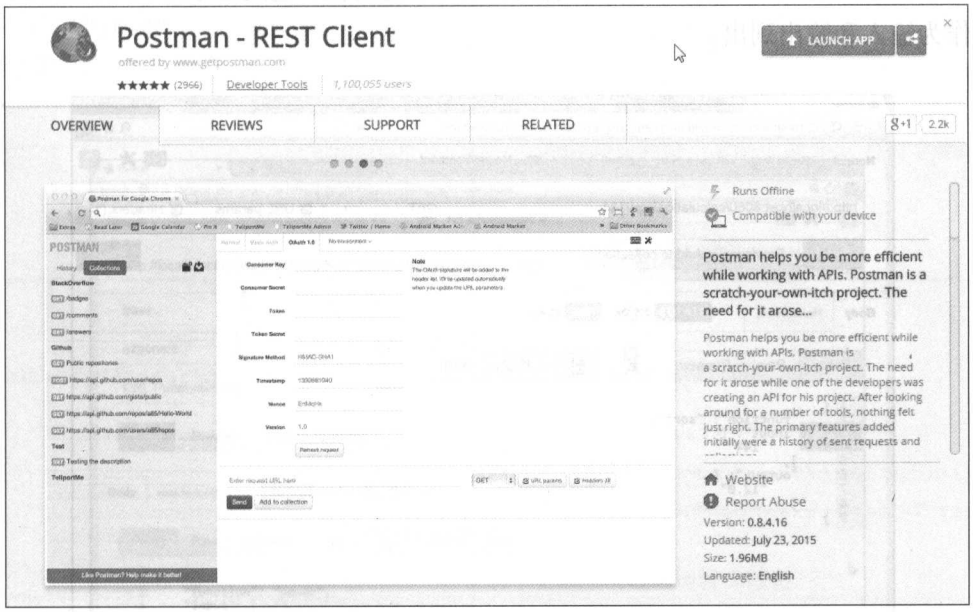
用 Java 命令执行

要生成 JAR，请执行 mvn clean 来打包 Maven 目标。它在 target 目录中创建 JAR 文件，然后可以使用如下命令执行 JAR:

```
java -jar target/restsample-1.0-SNAPSHOT.jar
```

使用 Postman Chrome 扩展测试 REST API

本书使用 Chrome 的 Postman-REST 客户端扩展来测试我们的 REST 服务。我们可以使用 `https://chrome.google.com/webstore/detail/postman-rest-client/fdmmgilgnpjigdojojpjoooidkmcomcm` 下载的 0.8.4.16 版本。此扩展不再是可搜索的，但可以使用给定的链接将其添加到 Chrome 中。此外可以使用 Postman Chrome 应用程序或任何其他 REST 客户端来测试示例 REST 应用程序。



Postman-REST 客户端 Chrome 扩展

一旦安装了 Postman-REST 客户端，就可以测试第一个 REST 资源。我们从开始菜单或快捷方式启动 Postman-REST 客户端。

在默认情况下，嵌入式的 web 服务器在端口 8080 上启动。因此，我们需要使用 http://localhost:8080/<resource>URL 来访问示例 REST 应用程序。例如：http://localhost:8080/calculation/sqrt/144。

一旦 web 服务器启动，就可以在 Calculation REST URL 中键入 sqrt 和值 144 作为路径参数，可以在下图中看到它。在 Postman 扩展的 URL 输入字段（在这里输入请求 URL）中输入此 URL。默认情况下，请求方法是 GET。我们把此默认值作为请求方法使用，因为我们也编写了 REST 式的服务来为 GET 请求方法提供服务。

一旦准备好上面提到的作为输入的数据，就可以通过单击 **Send** 按钮提交该请求。可以在下图中看到，示例 REST 服务返回响应代码 200。你能在下图中找到 **Status** 标签以看到 **200 OK** 代码。成功的请求也返回 Calculation 资源的 JSON 数据，如屏幕截图中的 Pretty 页签所示。返回的 JSON 显示 sqrt，它是 function 键的值。它还显示 144 和 12.0，它们分别作为输入和输出列出。

用Postman测试Calculation (sqrt)资源

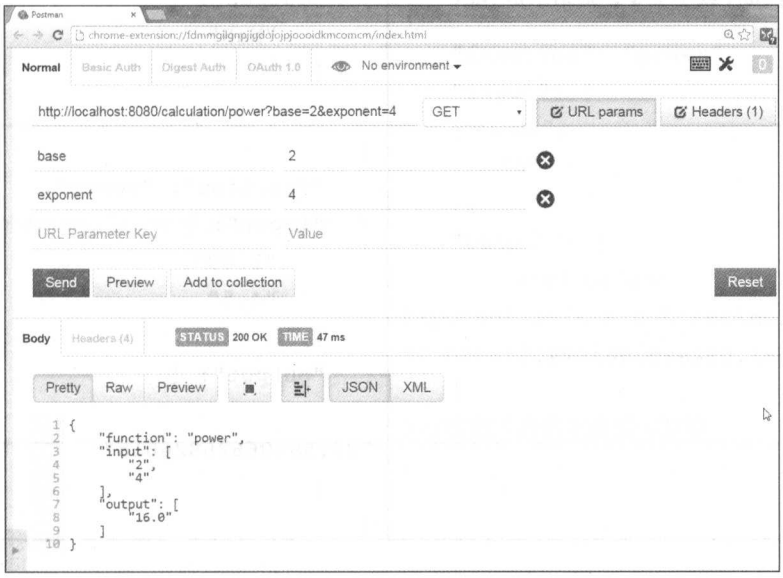
同样，我们也可以测试示例 REST 服务的 power 函数。在 Postman 扩展中输入以下数据：

- **URL**：`http://localhost:8080/calculation/power?base=2&exponent=4`
- **Request method**：`GET`

在这里，我们分别传入值 2 和 4 作为底数和指数的请求参数，它将返回下面的屏幕截图中所示的 **200** 的响应状态和以下 JSON。

返回的 JSON:

```
{
    "function": "power",
    "input": [
        "2",
        "4"
    ],
    "output": [
        "16.0"
    ]
}
```



使用Postman测试Calculation (power) 资源

更多的正向测试场景

在下面的表中，所有 URL 都以 `http://localhost:8080` 开头。

URL	输出 JSON
`/calculation/sqrt/12344.234`	`{ "function": "sqrt", "input": [ "12344.234" ], "output": [ "111.1046083652699" ] }`
`/calculation/sqrt/-9344.34` `Math.sqrt` 函数的特别场景： • 如果参数是 NaN 或小于零，则结果是 NaN	`{ "function": "sqrt", "input": [ "-9344.34" ], "output": [ "NaN" ] }`
`/calculation/power?base=2.09&exponent=4.5`	`{ "function": "power", "input": [ "2.09", "4.5" ], "output": [ "27.58406626826615" ] }`

续表

URL	输出 JSON

/calculation/power?base=-92.9&exponent=-4	{
	"function": "power",
	"input": [
	"-92.9",
	"-4"
	],
	"output": [
	"1.3425706351762353E-8"
	]
	}

反向的测试场景

同样，可以还执行一些反向的场景，如下表所示。此表中的所有 URL 都以 http://localhost:8080 开头。

URL	输出 JSON

/calculation/power?base=2a&exponent=4	{
	"function": "power",
	"input": [
	"2a",
	"4"
	],
	"output": [
	"Base or/and Exponent is/are not set to numeric value."
	]
	}//底数或指数未设置为数值

续表

URL	输出 JSON

/calculation/power?base=2&exponent=4b	{
	"function": "power",
	"input": [
	"2",
	"4b"
	],
	"output": [
	"Base or/and Exponent is/are not set to numeric value."
	]
	}//底数或指数未设置为数值

/calculation/power?base=2.0a&exponent=a4	{
	"function": "power",
	"input": [
	"2.0a",
	"a4"
	],
	"output": [
	"Base or/and Exponent is/are not set to numeric value."
	]
	}//底数或指数未设置为数值

/calculation/sqrt/144a	{
	"function": "sqrt",
	"input": [
	"144a"
	],
	"output": [
	"Input value is not set to numeric value."
	]
	}//输入值未设置为数值

续表

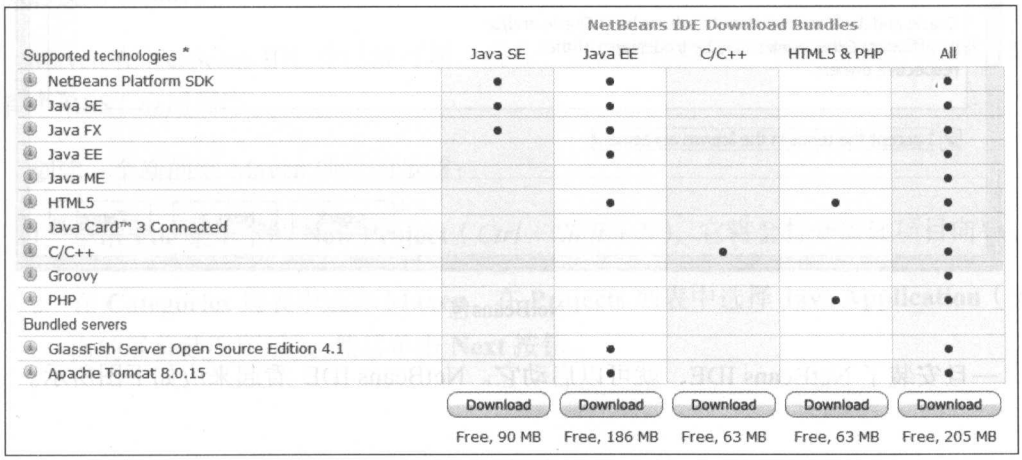
URL	输出 JSON

/calculation/sqrt/144.33$	{
	"function": "sqrt",
	"input": [
	"144.33$"
	],
	"output": [
	"Input value is not set to numeric value."
	]
	}//输入值未设置为数值

NetBeans IDE 安装和设置

NetBeans IDE 是免费并开放源码的，还有一个大的用户社区。可以从它的官方网站 https://netbeans.org/downloads/下载 NetBeans IDE。

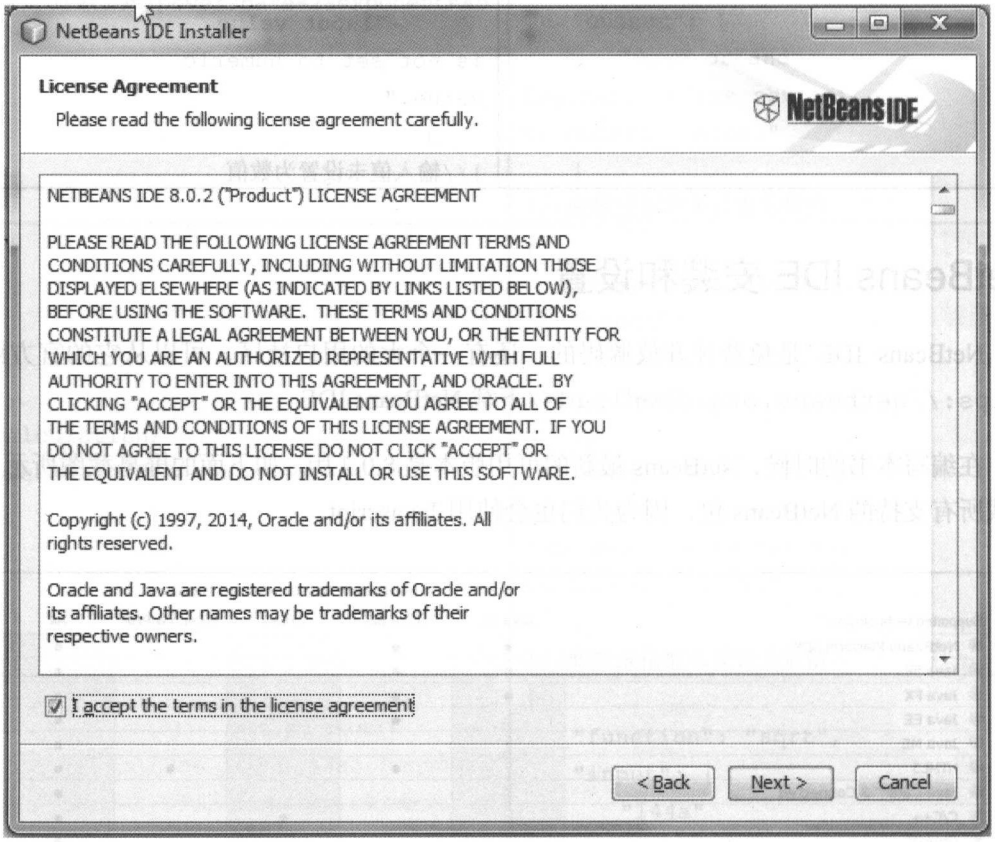
在编写本书的时候，NetBeans 最新的可用版本是 8.0.2 版。如下面的屏幕截图所示，请下载所有支持的 NetBeans 包，因为我们也会使用 Javascript。



NetBeans包

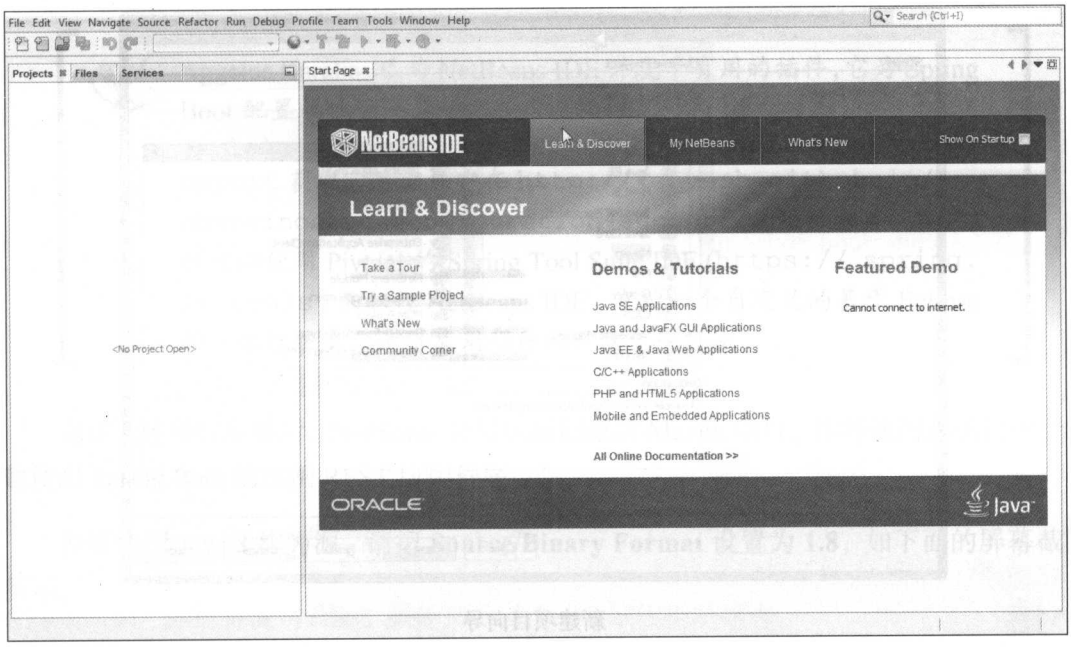
下载安装文件后，执行安装程序文件。接受使用协议，如下面的屏幕截图所示，并按照剩余步骤来安装 NetBeans IDE。Glassfish 服务器和 Apache Tomcat 是可选的。

安装和运行所有的 NetBeans 包需要 JDK 7 或更高版本的 JDK。可以从 `http://www.oracle.com/technetwork/java/javase/downloads/index.html` 下载一个独立的 JDK 8。



NetBeans包

一旦安装了 NetBeans IDE，就可以启动它。NetBeans IDE 看起来应如下图所示。



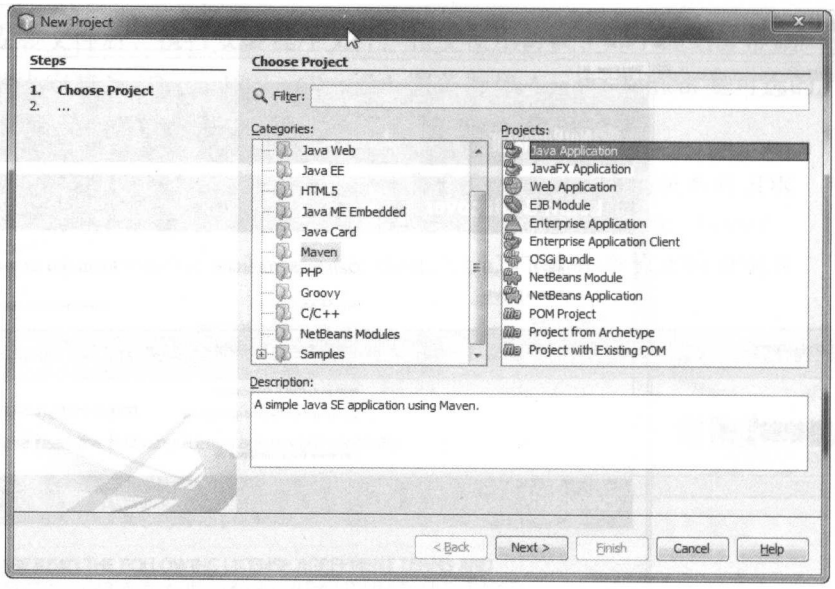
NetBeans 起始页

Maven 和 Gradle 两者都是 Java 构建工具，它们把依赖库添加到项目、编译你的代码、设置属性、建立归档文件，或执行多个相关的操作。Spring Boot 或 Spring Cloud 都同时支持 Maven 和 Gradle 构建工具。然而，在本书中我们将使用 Maven 构建工具。如果你喜欢，请随意使用 Gradle。

Maven 在 NetBeans IDE 中已经可用。现在，我们可以启动一个新的 Maven 项目来构建首个 REST 应用程序。

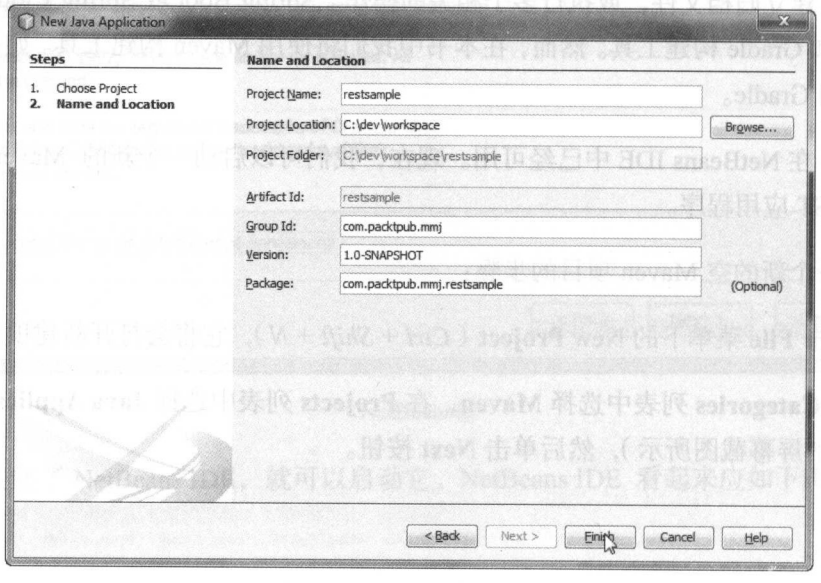
创建一个新的空 Maven 项目的步骤：

1. 单击 **File** 菜单下的 **New Project**（*Ctrl* + *Shift* + *N*），它将会打开新建项目向导。
2. 在 **Categories** 列表中选择 **Maven**，在 **Projects** 列表中选择 **Java Application**（如下面的屏幕截图所示），然后单击 **Next** 按钮。



新建项目向导

3. 现在，输入项目名称 `restsample`。此外，输入其他属性，如下面的屏幕截图所示。当所有必填字段都输入完成后，单击 **Finish** 按钮。

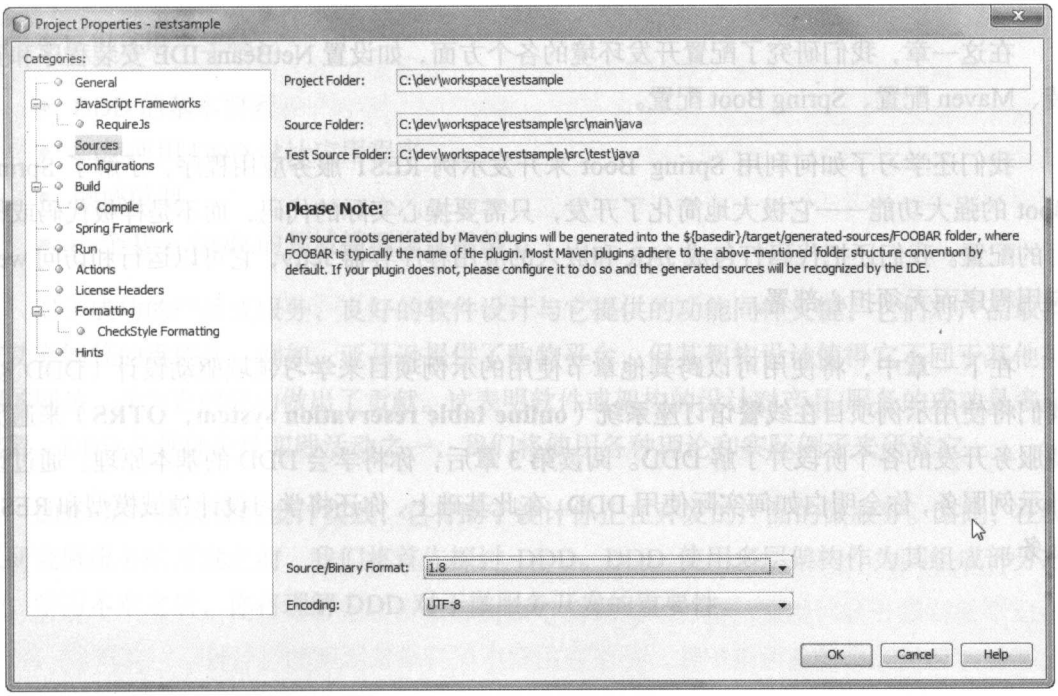


NetBeans Maven项目属性

Aggelos Karalias 已为 NetBeans IDE 开发了有用的插件，它为 Spring Boot 配置属性提供自动完成功能支持，可在 https://github.com/keevosh/nbspringboot-configuration-support 获取。可以从它在 http://keevosh.github.io/nbspringboot-configuration-support/的项目主页下载它。还可以使用 Pivotal 的 Spring Tool Suite IDE (https:// spring.io/tools) 来取代 NetBeans IDE。它是一个自定义的基于 Eclipse 的一体化版本，使得应用程序的开发变得容易。

完成上述所有步骤后，NetBeans 会显示新创建的 Maven 项目。你将使用此项目用于创建使用 Spring Boot 的示例 REST 应用程序。

若要使用 Java 8 作为源，请把 **Source/Binary Format** 设置为 **1.8**，如下面的屏幕截图所示。



NetBeans Maven项目属性

参考资料

- **Spring Boot**：http://projects.spring.io/spring-boot/
- 下载 **NetBeans**：https://netbeans.org/downloads
- **Representational State Transfer (REST)**：Roy Thomas Fielding 博士学位论文《Architectural Styles and the Design of Network-based Software Architectures》的第 5 章——https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
- **REST**：https://en.wikipedia.org/wiki/Representational_state_transfer
- **Maven**：https://www.apache.org/
- **Gradle**：http://gradle.org/

小结

在这一章，我们研究了配置开发环境的各个方面，如设置 NetBeans IDE 安装程序和安装、Maven 配置、Spring Boot 配置。

我们还学习了如何利用 Spring Boot 来开发示例 REST 服务应用程序，了解了 Spring Boot 的强大功能——它极大地简化了开发，只需要操心实际的代码，而不是样板代码或你写的配置。我们还把代码打包成 JAR 和嵌入的应用程序容器 Jetty，它可以运行和访问 web 应用程序而无须担心部署。

在下一章中，将使用可以跨其他章节使用的示例项目来学习领域驱动设计（DDD）。我们将使用示例项目**在线餐馆订座系统（online table reservation system，OTRS）**来遍历微服务开发的各个阶段并了解 DDD。阅读**第 3 章**后，你将学会 DDD 的基本原理。通过设计示例服务，你会明白如何实际使用 DDD。在此基础上，你还将学习设计领域模型和 REST 服务。

3 领域驱动设计

本章通过引用一个示例项目为其余的章节定基调。从这里开始，将用这个示例项目来解释不同的微服务概念。本章使用此示例项目驱动不同的功能组合和领域服务或应用程序来解释**领域驱动设计（domain driven design，DDD）**。它将帮助你了解 DDD 及其实际用法的基础知识。你还将使用 REST 服务来学习设计领域模型的概念。

本章包含以下主题：

- DDD 的基本要素
- 如何使用 DDD 设计应用程序
- 领域模型
- 一个基于 DDD 的领域模型设计示例

对于成功的产品或服务，良好的软件设计与它提供的功能同样关键，它们对产品取得成功有同等的重要性。例如，亚马逊提供了购物平台，但其架构设计使得它不同于其他类似的网站，并为它的成功做出了贡献。这表明软件或架构的设计对产品/服务的成功是多么重要。DDD 是软件设计实践活动之一，我们将使用各种理论和实际例子来研究它。

DDD 是一种关键的设计实践，它有助于设计你正在开发的产品的微服务。因此，在深入研究微服务的开发之前，我们将首先探讨 DDD。DDD 使用多层架构作为其组成部分之一，学习本章之后，你将理解 DDD 对于微服务开发的重要性。

领域驱动设计基本原理

企业或云应用程序是用来解决业务问题和其他现实世界的问题的。若没有所在领域的知识，就不能解决这些问题。例如，如果你不懂股票交易和其运作的原理，你就不能为金融系统，如网上股票交易提供一个软件解决方案。因此，拥有领域知识是解决问题的一个必备条件。现在，如果你想要提供一个使用软件或应用程序的解决方案，那么你需要领域知识的帮助来设计它。当我们将领域和软件设计相结合时，就产生了一种称为 DDD 的软件设计方法。

当我们开发软件来实现提供某个领域的功能的现实世界场景时，我们就创建了该领域的一个模型。模型是该领域的一种抽象或蓝图。

Eric Evans 在其于 2004 年出版的《领域驱动设计：处理位于软件核心的复杂性》（*Domain-Driven Design: Tackling Complexity in the Heart of Software*）这本书中首创了 DDD 这个术语。

虽然设计这种模型不像火箭科学那么艰难，但也需要付出大量的努力，需要领域专家的提炼和输入。它是软件设计师、领域专家和开发人员的集体工作。他们对信息进行组织，将其分成较小的部分，从逻辑上对它们进行分组并创建模块。每个模块可以单独处理，也可以使用类似的方法划分。我们可以遵循这一过程，直到到达单元级别或我们不能将它进一步划分为止。一个复杂的项目可能会有多次这样的迭代，而同样一个简单的项目可能只有单个的迭代。

一旦定义好了模型，并将其良好地记录在文档中，就可以转到下一个阶段——代码设计。所以，在这里我们有一个软件设计——领域模型和代码设计——和领域模型的代码实现。领域模型提供了高层次的解决方案（软件/应用程序）的架构，而代码实现作为一种能工作的模型给了领域模型生命。

DDD 使设计和开发人员团结合作。它持续不断地开发软件，同时根据从开发人员收到的反馈保持设计与时俱进的能力。它解决了敏捷和瀑布方法带来的局限性之一，即让软件包括设计和代码都具备可维护性，并使应用程序保持最低限度的可行性。

设计驱动的开发从初始阶段就涉及一个开发人员，他参加在建模过程中软件设计师与领域专家讨论领域的所有会议。这为开发人员提供合适的平台来了解领域，并提供机会来分享领域模型实现的早期反馈。它将消除在软件开发后期出现的股东等待可交付结果时的瓶颈。

组成部分

这部分描述 DDD 所采用的普遍存在的语言，以及之所以需要它的原因，用于模型驱动设计的不同模式和多层架构的重要性。

普遍存在的语言

正如我们所看到的，设计模型是软件设计师、领域专家和开发人员集体的工作，因此，它需要使用共同的语言来交流。DDD 使得有必要使用共同的语言，并使用普遍存在的语言。领域模型在其关系图、说明、陈述、演讲和会议中使用普遍存在的语言。这将消除误解、曲解和他们之间的沟通鸿沟。

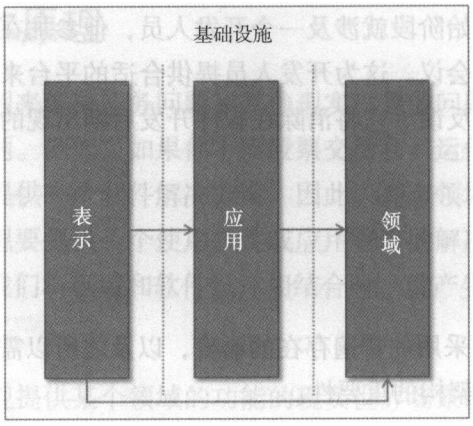
统一建模语言（Unified Model Language，UML）在创建模型时广泛使用，很流行。但这种语言也存在一些限制，例如，当你有一份绘制了成千上万个类的文件时，很难表示类之间的关系，也很难在理解它们的抽象性的同时理解它的含义。此外 UML 关系图也不会表示模型的概念以及对象都应该做些什么。

还有其他沟通领域模型的方法，如——文档、代码，等等。

多层架构

多层架构是 **DDD** 的一个通用解决方案。它包含四个层次：

1. 表示层或用户界面（User Interface，UI）。
2. 应用层。
3. 领域层。
4. 基础设施层。



分层架构

在此可以看到只有领域层负责领域模型，而其他层都是与其他组件，例如用户界面、应用程序逻辑相关的。这种分层的架构非常重要，它使领域相关的代码与其他层分开。

在这种多层架构中，每层都包含其各自的代码，这有助于实现松耦合并避免把不同层的代码相混合，还有助于产品/服务的长期可维护性和增强功能的方便性。因为如果修改的目的只是针对各自的层，那么其中一层代码的修改不会对其他组件造成影响。使用多层架构，每一层都可以很容易地用另一个实现来切换。

表示层

此层表示 UI，并为交互和信息显示提供用户界面。这一层可能是一个 web 应用程序、移动应用或使用你的服务的第三方应用程序。

应用程序层

此层负责应用程序逻辑。它维护并协调产品/服务的整体工作流，它不包含业务逻辑或 UI，它可以保持应用程序对象的状态，如执行中的任务的状态。例如，你的产品 REST 服务将是应用层的一部分。

领域层

领域层是非常重要的一层，因为它包含领域信息和业务逻辑。它保持业务对象的状态，

它持久化业务对象的状态并将这些持久化的状态与基础设施层进行通信。

基础架构层

这一层向所有其他层提供支持，并负责其他层之间的通信。它包含由其他层使用的支持库，它还实现业务对象的持久化。

为了了解不同层的相互作用，让我们以一家餐馆的餐桌预订为例说明。最终用户使用UI做出预订一个餐桌的请求，用户界面将该请求传递到应用程序层。应用程序层从领域层提取领域对象，如餐馆、带有日期的餐桌，领域层从基础设施获取这些现有的持久化对象，并调用相应的方法进行预订并把它们持久化后返回给基础设施层。当领域对象被持久化后，应用层就给最终用户显示预订确认消息。

领域驱动设计的工件

在DDD中存在用于表示、创建和获取领域模型的不同工件。

实体

存在某些类别的可确定的对象，它们保持产品或服务的整体状态。这些对象不是（**NOT**）由其属性定义，而是由其身份和线程的连续性定义的。这些对象被称为实体。

这听起来很简单，但它承载着复杂性。你需要了解怎么定义实体。以餐馆订座系统为例，如果有`restaurant`类，包含餐馆名称、地址、电话号码以及建立数据等属性，可以取`restaurant`类的两个实例，它们不可使用餐馆名称来确定，因为可能还有其他餐馆具有相同名称。同样，如果通过任何其他单个属性去确定，也不会找到任何可以确定唯一的餐馆的属性。如果两个餐馆所有的属性值都相同，那么这两个餐馆是相同的，可以彼此互换。尽管如此，这些仍然是不同的实体，因为它们都具有不同的引用（内存地址）。

相反，让我们来看一个美国公民的类。每个公民都有他自己的社会安全号码。这个数字不但唯一，而且在公民的一生中不变，并保证连续性。这个公民对象将存在于内存中，会被序列化，并将会从内存中删除并且存储在数据库中。即使在该公民已去世后，它仍然存在。只要系统存在，它就将保留在系统中。公民社会安全号码保持不变，不论它的表示形式如何。

因此，在产品中创建实体是指创建标识符。所以，现在把标识符分配给前面示例中的任何一个餐馆，然后使用诸如餐馆名称、成立日期、街道等属性的组合或者添加标识符，如 `restaurant_id` 来标识它。基本规则是任何两个标识符都不能相同。因此，当我们为任何实体引入标识符时，我们需要确保这一点。

为对象创建一个唯一的标识符有不同的方式，如下所述：

- 使用表中的主键。
- 使用领域模块自动生成的 ID。领域程序生成标识符，并将它分配给在不同层间被持久化的对象。
- 少数现实生活中的对象本身就自带用户定义标识符。例如每个国家都有其自己的国际长途电话拨号国家/地区代码。
- 一个属性或属性的组合可用于创建一个标识符，正如前面对 `restaurant` 对象的解释。

实体对于领域模型是非常重要的，因此，应在建模过程的初始阶段就定义它们。当一个对象可以通过其标识符而不是它的属性来确定时，表示这些对象的类应该有一个简单的定义，并且应维护生命周期连续性和身份。对此类中具有相同属性值的对象进行标识势在必行。如果查询每个对象，一个既定的系统应返回唯一的结果。负责维护模型的设计师必须对同样的事物意味着什么加以定义。

值对象

实体具有一些特征，如身份、连续性的线程以及不能确定其身份的属性。**值对象（Value objects，VO）**只是具有属性，而没有概念上的身份。最好的做法是把值对象保持为不可变的对象。如果可能，你甚至也应该保持实体对象不可变。

实体概念可能偏要你把所有对象作为实体来保持，内存或数据库中的唯一可识别对象具有生命周期的持续性，但每个对象都必须有一个实例。现在，假设你正在把客户作为实体对象来创建。每个客户对象都表示餐馆的顾客，这个对象不能用于下其他顾客的订单。如果有数以百万计的客户使用此系统，这可能在内存中创建数以百万计的客户实体对象。在系统中，不止存在数以百万计的唯一可识别的对象，而且每个对象都被追踪。追踪和创建身份都是复杂的。创建和追踪这些对象需要高度可靠的系统，这个系统不但很复杂，而

且重度消耗资源。这可能会导致系统性能下降。因此，必须使用值对象而不是使用实体。原因将在接下来的几段中解释。

应用程序不总是需要有一个可识别的客户对象并加以追踪。有些情况下，只需要有领域元素的一些或全部属性。这些都是可以由应用程序使用值对象的情况，它使事情变得简单并提高了性能。

由于不存在身份，值对象可以被方便地创建和销毁。这简化了设计——它使得值对象在没有其他对象引用它们时可用于垃圾回收。

让我们讨论一下值对象的不变性。值对象应被设计和编码为不可变的。一旦创建了它们，它们绝不应在其生命周期中被修改。如果你需要此 VO 的一个不同的值，或其任何对象，那么只需创建一个新的值对象，但不要修改原始的值对象。在此，不变性从**面向对象设计（object-oriented programming，OOP）**继承了所有意义。当且仅当值对象是不可变的时，它才可以被共享和使用而不会影响其完整性。

常见的问题

1. 一个值对象可以包含另一个值对象吗？

是的，可以。

2. 一个值对象可以引用另一个值对象或实体吗？

是的，可以。

3. 可以使用不同的值对象或实体的属性来创建一个值对象吗？

是的，可以。

服务

创建领域模型时可能会遇到一些情况，其中的行为不能与任何对象明确关联。这些行为可以容纳在服务对象中。

在领域建模过程中，普遍存在的语言可帮助你确定具有不同属性和行为的不同对象、身份或者值对象。在创建领域模型的过程中，你可能会发现不同的不属于任何特定对象的

行为或方法。这些行为是重要的，所以不能忽视，也不能把它们添加到实体或值对象中。把行为添加到不属于它的对象中会破坏此对象。请记住，那些行为可能影响各种对象。使用面向对象编程可以将它们附加到一些对象上，这就是所谓的服务。

服务在技术框架中是常见的。这些也都用在 DDD 的领域层中。服务对象不具有任何内部状态，它的唯一目的是提供针对领域的行为。服务对象提供不能与特定实体或值对象相关联的行为，可能会为实体或值对象提供一个或多个相关的行为。这是在一个领域模型中显式定义的服务的做法。

在创建服务时，你需要勾选所有以下各点：

- 在实体和值对象上执行的服务对象的行为，但它不属于实体或值对象
- 服务对象的行为状态不会被保持，因此它们是无状态的
- 服务是领域模型的一部分

服务可能还存在于其他层中。保持领域层服务的隔离非常重要，它消除复杂性并保持设计解耦。

让我们看一个例子，一个餐馆老板想查看每月的餐桌预订的报告。在这种情况下，他将以管理员的身份登录，并提供所需的输入的字段，如时间段，然后单击 **Display Report**（**显示报告**）按钮。

应用程序层将请求传递到拥有的报告和模板对象的领域层，这种请求带有一些参数，如报表 ID 等。报告使用模板和从数据库或其他来源获取数据来创建。然后，应用程序层向业务层传递包括报告 ID 的所有参数。此时，需要从数据库或其他来源获取模板基于此 ID 生成报告。此操作不属于报表对象或模板对象。因此用一个服务对象来执行此操作，以便从数据库中提取所需的模板。

聚合

聚合域模式与对象的生命周期相关，并定义所有权和边界。

当你使用任何应用程序在网上预订你最喜欢的餐馆的餐桌时，你不需要操心处理预订的内部系统和过程，比如搜索可用的餐馆，然后搜索给定的日期和时间可用的餐桌，等等。因此，可以说一个预订应用程序是其他对象和作品的聚合，它充当一个餐馆订座系统的所

有其他对象的根。

这个根应该是将对象集合结合在一起的实体，它也称为聚合根。这个根对象不把对任何内部对象的引用传递到外部世界，并保护对内部对象执行的更改。

我们需要明白为什么需要聚合。领域模型可以包含大量的领域对象，应用程序的功能和规模越大，其设计越复杂，它就会存在越多的对象。这些对象之间存在关系，一些可能有多对多的关系，另一些可能有一对多的关系，而其他的可能有一对一的关系。这些关系是由代码或数据库中的模型实现来实施的，以确保这些对象之间的关系保持完整。关系不只是单向的，它们也可以是双向的。这也可能增加复杂性。

设计师的任务是简化这些模型中的关系。一些关系可能存在于一个现实的领域中，但可能在领域模型中不需要。设计师需要确保领域模型中不存在这种关系。同样，这些约束也可以减少多样性。在许多对象都满足某个关系的情况下，可以用一个约束来做这项工作。双向关系也可能转化为单向关系。

无论对输入进行多少简化，你仍然会面对模型中的关系。这些关系需要有维护它们的代码。当一个对象被删除时，代码应该从其他地方删除对此对象的所有引用。例如，从一个表中删除一行记录时，需要对它以外键形式引用的所有地方都加以处理，以保持数据的一致性并维护其完整性。在数据发生更改时，不变量（规则）也需要被实施和维护。

关系、约束和不变量带来的复杂性需要在代码中进行有效处理。我们利用由称为根的单一实体表示的聚合来解决这个问题，根与数据更改时用来保持一致性的一组对象相关联。

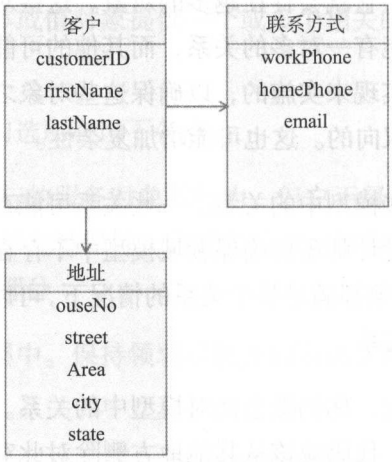
根是唯一可以从外部访问的对象，所以此根元素充当把内部对象与外部世界分离的边界的大门。根可以引用一个或更多的内部对象，并且这些对象内可以引用其他内部对象，而这些内部对象与根之间可能有关系，也可能没有关系。然而，外部对象也可以引用根，但不能引用任何内部对象。

聚合可确保数据的完整性并实施不变量。外部对象不能更改内部对象，它们只能改变根。然而，它们可以使用根，通过调用公开的操作，在此对象内部做出更改。根应该把所需的内部对象的值传递给外部对象。

如果一个聚合对象存储在数据库中，那么查询应该只返回此聚合对象。当对象内部链接到聚合根时，应该用遍历来返回它。这些内部的对象也可能会引用其他聚合。

聚合根实体保存其全局身份并保存其各实体内的本地身份。

在餐馆订座系统中，聚合的一个简单的例子就是客户。客户可以接触到外部对象，并且其根对象包含其内部对象的地址和联系信息。当发出请求时，就可以把诸如地址等内部对象的值对象传递给外部对象。



客户作为一个聚合

存储库

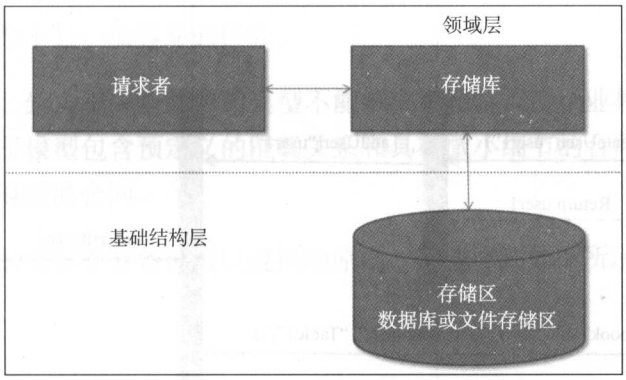
在一个领域模型中，在某一给定的时间，可能存在多个领域对象。每个对象都可能具有从创建到去除或持久的生命周期。每当任何领域操作需要一个领域对象时，它都应该有效地获取所请求的对象的引用。如果不把所有可用的领域对象都保存在一个中心的对象中，并由这个中心对象保存所有对象的引用并负责返回请求的对象的引用，这个任务将变得非常困难。这个中心的对象称为存储库（repository）。

存储库是与基础设施如数据库或文件系统进行交互的一个地方。存储库对象是领域模型中与诸如数据库、外部来源等存储区（storage），以检索持久化的对象进行交互的一部分。当存储库接收到对某个对象的引用请求时，它将返回现有对象的引用。如果存储库中不存在请求的对象，那么它就从存储区中获取对象。例如，如果你需要某个客户，你将查询存储库对象，请求它提供 ID 为 31 的客户。如果这个客户对象已经在存储库中，存储库就将提供请求的客户对象，如果不是这样，就将查询诸如数据库之类的持久化存储，把它取出

并提供它的引用。

使用存储库的主要优点是具有用来获取对象的一致方法，其中请求者不需要直接与诸如数据库的存储区进行交互。

存储库可以从各种存储区类型查询对象，这些存储区类型包括一个或多个数据库、文件系统或工厂资料库，等等。在这种情况下，存储库中可能也有指向不同的对象类型或类别的不同来源的策略。



存储库对象工作流

如上图所示，存储库与基础设施层交互，此接口是领域层的一部分。请求者可能属于领域层或应用层。存储库帮助系统来管理领域对象的生命周期。

工厂

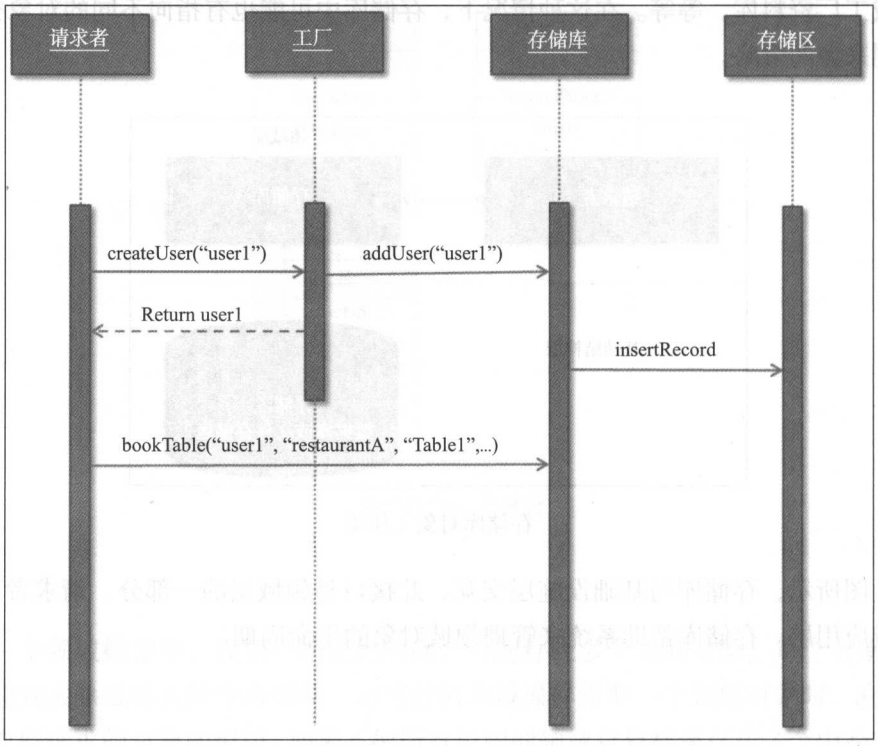
当一个简单的构造函数并不足以创建对象时，就需要用到工厂。它有助于创建复杂的对象或一个涉及其他相关对象的创建的聚合。

工厂也是领域对象生命周期的一部分，因为它负责创建它们。工厂和存储库以某种方式彼此相关，因为两者都引用领域对象。工厂引用新创建的对象，而存储库从内存中或从外部存储器中返回已存在的对象。

让我们看看使用一个用户创建过程的应用程序的控制是如何流动的。假设一个用户注册用户名 user1。此用户创建过程首先与工厂进行交互，创建名称 user1，然后使用存储库将其缓存在领域中，存储库还将它存储在持久性存储区中。

当同一用户再次登录时，调用就转移到存储库中来获取一个引用。这使用存储区来加载此引用并将其传递给请求者。

然后请求者可能使用此 user1 对象在指定餐馆和指定的时间预订餐桌。这些值作为参数进行传递，并使用存储库在存储区中创建餐桌的一条预订记录。



存储库对象工作流

工厂可以使用某种面向对象的编程模式，如工厂或抽象工厂模式之一来创建对象。

模块

模块是分离相关的业务对象的最好方法。这些都是最适合于领域对象的规模更大的大型项目。对于最终用户，把领域模型划分为模块并设置这些模块之间的关系是有道理的。一旦理解了模块和它们之间的关系，就会看到领域模型更宏观的画面，并且更容易进一步钻研和理解模型。

模块还有助于产生很高的内聚或保持低耦合的代码。普遍存在的语言可以用于命名这些模块。对于餐馆订座系统，我们可以拥有不同的模块，如用户管理、餐馆和餐桌、分析和报告，以及评论，等等。

战略设计和原则

企业模型通常都非常庞大和复杂。它可能分布在组织的不同部门之间，每个部门都可能有单独的领导团队，所以在一起工作和设计可能产生困难和协调问题。在这种情况下，维持完整的领域模型不是一项容易的任务。

在这种情况下，致力于一个统一的模型不能解决问题，大型企业模型需要被划分为不同的子模型。这些子模型包含预定义的准确关系和具有微小细节的合同。每个子模型都必须无一例外地保持既定的合同。

保持领域模型的完整性有各种可以遵循的原则，这些原则如下所示。

- 有界上下文
- 持续集成
- 上下文映射
  - 共享内核
  - 客户-供应商
  - 顺从者
  - 反腐层
  - 各自的方法
  - 开放主机服务
  - 精馏

有界上下文

当你有不同的子模型时，如果所有子模型都被结合在一起，代码是难以维护的。你需要有一个可以分配给单个团队的小型模型。你可能需要收集相关的元素，并将它们组合起来。上下文通过应用此组条件来保持并维护为其各自的子模型定义的领域术语的含义。

这些领域的术语定义创建上下文边界模型的范围。

有界上下文看起来非常类似于在上一节中了解到的模块。事实上，模块是有界的上下文的一部分，它定义了子模型产生并被开发的逻辑框架。而模块组织领域模型的元素，并可以在设计文档和代码中看见。

现在，作为一个设计师，你需要保持每个子模型的明确和一致性。以这种方式，可以在不影响其他子模型的情况下独立地重构每个模型。这使软件设计师可以在任何时点灵活地完善和改进它。

现在看看餐桌预订示例。当你开始设计系统时，你就会预见到客人将访问该应用程序，并将请求在所选的餐馆、日期和时间保留餐桌。然后在此基础上，还有把预订信息通知餐馆的后端系统。同样，餐馆会根据餐桌预订来更新他们的系统，因为餐桌可以也由餐馆自己预定。所以，当你更细致地查看这个系统时，可以看到两个领域模型：

- 在线餐馆订座系统
- 离线餐馆管理系统

每个领域模型都有其有界的上下文，需要确保它们之间的接口工作正常。

持续集成

当你进行开发时，代码散布在很多团队中，并使用各种技术，此代码可以被组织成不同的模块，并且各子模型都有适用的有界上下文。

这种开发可能会带来某种程度的重复代码、代码冲突或破坏有界上下文方面的复杂性。这种情况的发生不仅因为代码和领域模型的规模庞大，而且因为有其他因素的影响，如团队成员的变化，新成员或缺少有据可查的模型来命名其中的一小部分。

当使用 DDD 和敏捷方法来设计和开发系统时，在开始编码之前，领域模型不是完全设计好的，并且随着时间的推移，经历一段时间的不断改进和完善，领域模型及其元素都发生了进化。

因此，集成在持续进行，这是目前开发的关键原因之一，所以它发挥着非常重要的作用。在持续集成中，经常合并代码以避免造成任何破坏和领域模型的问题。合并的代码不

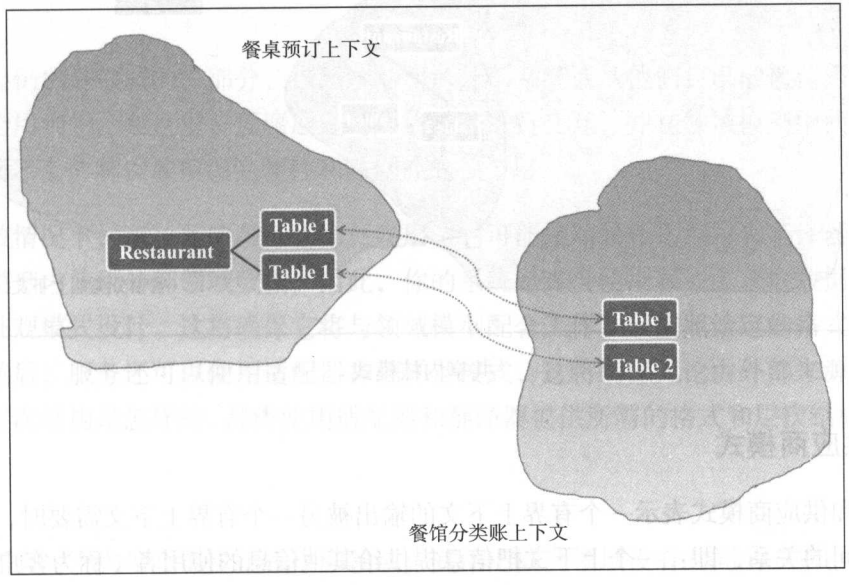
仅得到部署，而且还被定期测试。市场上有能够在预定的时间合并、构建和部署代码的各种持续集成工具。现在，各个组织更加注重持续集成的自动化。Hudson、TeamCity 和 Jenkins CI 是现在可用来进行持续集成的几个流行的工具。Hudson 和 Jenkins CI 是开放源代码工具，而 TeamCity 是一个专有的工具。

每个生成版本都具有附加的测试套件来验证模型的一致性和完整性。测试套件从物理的角度定义模型，而 UML 则在逻辑上定义模型。它告诉你有关任何错误或意外的结果，需要更改代码来改正它们，它还有助于及早发现领域模型中的错误和异常。

上下文映射

上下文映射帮助你理解一个大型企业应用程序的整体情况。它显示在企业模型中有多少有界上下文存在，以及它们如何相互关联。因此我们可以把任何解释了有界上下文和它们之间的关系的关系图或文档都称为上下文映射。

上下文映射能帮助所有团队成员，无论他们是在同一个团队还是在不同的团队，以各种部件（有界上下文或子模型）和关系的形式了解高层次企业模型。这使每个人都能清楚地知道某个人执行的任务，并允许他提出模型的完整性有关的任何关注点/问题。



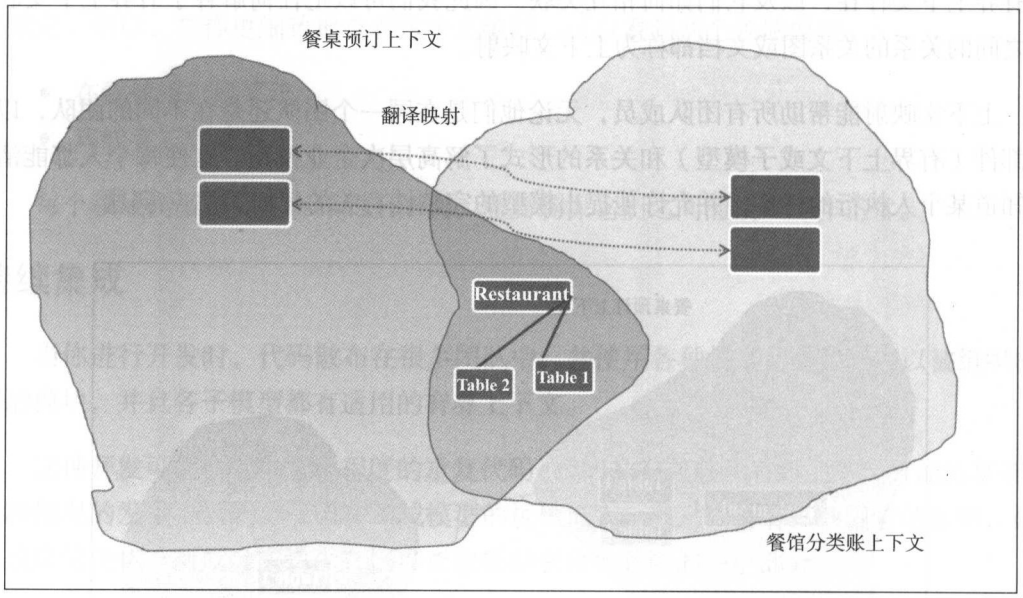
上下文映射示例

上下文映射的关系图是上下文映射的一个示例。在这里，**Table1** 和 **Table2** 二者都既出现在**餐桌预订上下文**中，又出现在**餐馆分类账上下文**中。有趣的事情是，**Table1** 和 **Table2** 在每个有界上下文中都有各自的概念。在这里，用普遍存在的语言把有界上下文命名为餐桌预订和餐馆分类账。

在下面的部分，我们将探索一些可以用来定义上下文映射中的不同上下文之间通信的模式。

共享内核模式

正如其名字所示，一个有界上下文与另一个有界上下文共享一个部分。正如你可以看到的，下面的 **Restaurant** 实体正在**餐桌预订上下文**和**餐馆分类账上下文**之间共享。



共享内核模式

客户和供应商模式

客户和供应商模式表示一个有界上下文的输出被另一个有界上下文需要时，两个有界上下文之间的关系，即：一个上下文把信息提供给其他信息的使用者（称为客户）。

在现实的例子中，一个汽车经销商不能出售汽车，除非汽车制造商提供了这些汽车。因此，在此领域模型中，汽车制造商是供应商，而经销商是客户。这种关系建立了一种客户和供应商的关系，因为一个有界上下文（汽车制造商）的输出（汽车）是另一个有界上下文（经销商）所必需的。

在这里，客户和供应商团队应定期开会，建立一份合同，并形成正确的互相交流的协议。

顺从者模式

这种模式类似于客户和供应商模式，一方需要提供合同和信息，而另一方需要使用它。在这里，不同于有界上下文，实际参与的团队有上游下游的关系。

此外，上游团队因为缺乏动机而不对下游团队提供需要的信息。因此，下游团队可能需要计划和处理绝不可能获得的项目。要解决这种情况，如果供应商提供的信息不足，任一客户团队都可以开发他们自己的模型。如果供应商提供的信息是真正有价值或部分有价值的，那么客户就可以利用可用来消费供应商提供的信息的接口或翻译器来处理客户自己的模型。

反腐层

反腐层仍然是领域的一部分，当系统需要从外部系统或从他们自己的遗留系统获取数据时，就会用到它。在这里，反腐层是与外部系统进行交互，并在领域模型中使用外部系统数据，而不会影响领域模型完整性和独创性的一个层。

大多数情况下，可以使用服务作为反腐层，它可能使用具备适配器和翻译器的外观模式在内部模型内使用外部领域数据。因此，你的系统始终会使用服务来获取数据。服务层可以使用外观模式设计。这将确保它将与领域模型配合工作，以按照给定的格式提供所需的数据。然后，服务还可以使用适配器和翻译器模式，这将确保无论由外部来源发送的数据格式和层次结构是怎样的，都将使用适配器和翻译器提供所需的格式和层次结构的服务。

独立方法

当你拥有一个大型企业应用程序和领域，其中不同的领域没有共同的元素，并且它由能独立开展工作的大型子模型组成时，对于最终用户，这仍然作为单个应用程序工作。

在这种情况下，设计师可以创建单独的没有相互关系的模型，并在它们上面开发一个小应用程序。这些小应用程序合并在一起时成为一个单独的应用程序。

工作单位内部网应用程序是这类应用程序中的一种，它提供如人力资源相关、问题跟踪器、运输或公司内部社交网络的各种小应用程序，设计师可以使用独立方法的模式来开发它。

若要集成使用独立模型开发的应用程序，这将具有非常大的挑战性和复杂性。因此，在实现这种模式之前，你应该三思而行。

开放主机服务

当两个子模型彼此交互时，就会使用翻译层。这个翻译层用于把模型与外部系统集成。当你有一个使用该外部系统的子模型时，这种模式工作正常。当这个外部系统被用于很多子模型，以删除多余和重复的代码时，因为你要为每个子模型的外部系统都编写一个翻译层，则需要开放主机服务。

开放主机服务使用对所有子模型的包装提供外部系统的服务。

精馏

正如你所知，精馏是净化液体的过程。同样，在 DDD 中，精馏是过滤掉不必要的信息，并只保留有意义的信息的过程。它可以帮助你识别核心领域和业务领域的基本概念，过滤掉一般的概念，直到你得到代码域的概念为止。

开发人员和设计师对核心领域进行设计、开发和实施时应对细节加以最高的关注，因为它是整个系统成功的关键。

在我们的餐馆订座系统示例中，它不是一个大型或复杂领域的应用程序，它的核心领域不难识别。核心领域在这里的存在是为了共享实时准确的在餐馆里的空置餐桌，并允许用户用不麻烦的过程来预订它们。

示例领域服务

让我们基于餐馆订座系统创建一个示例领域服务。如本章中所述，有效领域层的重

要性是成功的产品或服务的关键。基于领域层开发项目有更好的可维护性、高内聚性和解耦性。它们在业务需求变化时能够提供高可扩展性，并对其他各层的设计有较少的影响。

领域驱动开发基于领域，因此不推荐你使用首先开发 UI，紧接着开发其余的层，最后开发持久层的自顶向下的方法，也不推荐首先开发类似数据库的持久层，然后开发其余的层，最后开发 UI 的自底向上的方法。

首先使用本书描述的模式开发出一个领域模型，将使所有团队成员对功能都有清晰的认识，便于软件设计师来建立一个灵活的、可维护的、一致的系统，可以帮助组织以较低的维护成本来开发世界一流的产品。

在这里，你将创建提供添加和检索餐馆功能的一个餐馆服务。基于具体实现，你可以添加其他功能，如基于菜品或评级查找餐馆。

从实体开始。在这里，餐馆是我们的实体。因为每个餐馆都是唯一的，并有一个标识符。可以使用一个接口或一组接口在我们的餐馆订座系统中实现实体。理想情况下，如果采用接口隔离原则，你会使用一组接口，而不是一个单独的接口。

接口隔离原则（Interface Segregation Principle，ISP）：客户不应该被强迫去依赖于他们不使用的接口。

实体的实现

对于第一个接口，你可以有一个所有实体所需的抽象类或接口。例如，如果我们考虑 ID 和名称，对于所有实体，属性都将是常见的。因此，可以在你的领域层使用抽象类 `Entity` 作为实体的抽象：

```
public abstract class Entity<T> {

    T id;
    String name;

}
```

基于此，你也可以有另外一个继承 Entity 的抽象类：

```
public abstract class BaseEntity<T> extends Entity<T> {

    private T id;
    public BaseEntity(T id, String name) {
        super.id = id;
        super.name = name;

    }
    ... (getter/setter 和其他相关的代码)
}
```

基于前面的抽象，我们可以创建用于餐馆管理的 Restaurant 实体。

现在，因为我们正在开发的是餐馆订座系统，Table 是领域模型中另一个重要的实体。所以，如果我们采用聚合模式，restaurant 会作为一个根来工作，而 Table 将是 Restaurant 内部的实体。因此，Table 实体将始终可以使用 Restaurant 实体来访问。

可以用下面的实现创建 Table 实体，并且可以根据需要添加属性。只为了示范，下面使用基本的属性：

```
public class Table extends BaseEntity<BigInteger> {

    private int capacity;

    public Table(String name, BigInteger id, int capacity) {
        super(id, name);
        this.capacity = capacity;
    }

    public void setCapacity(int capacity) {
        this.capacity = capacity;
    }

    public int getCapacity() {
        return capacity;
```

```
    }
}
```

现在，我们可以实现如下所示的聚合器 Restaurant。在这里，只使用了基本的属性。可以根据需要添加任意多的属性，还可以添加你想要的其他功能：

```
public class Restaurant extends BaseEntity<String> {

    private List<Table> tables = new ArrayList<>();
    public Restaurant(String name, String id, List<Table> tables) {
        super(id, name);
        this.tables = tables;
    }

    public void setTables(List<Table> tables) {
        this.tables = tables;
    }

    public List<Table> getTables() {
        return tables;
    }
}
```

存储库的实现

现在，我们可以来实现在这一章中了解的存储库模式。开始时，你将首先创建两个接口 Repository 和 ReadOnlyRepository。ReadOnlyRepository 将被用于提供只读操作的抽象，而 Repository 抽象将用于执行所有类型的操作：

```
public interface ReadOnlyRepository<TE, T> {

    boolean contains(T id);

    Entity get(T id);

    Collection<TE> getAll();
}
```

基于该接口，我们可以创建会做额外的操作，如添加、删除和更新的存储库抽象：

```
public interface Repository<TE, T> extends ReadOnlyRepository<TE, T> {

    void add(TE entity);

    void remove(T id);

    void update(TE entity);
}
```

前面定义的存储库抽象可以用适合的方式来持久化你的对象。在持久性代码，即基础设施层的一部分所做的改变不会影响领域层代码，因为合同和抽象是由领域层定义的。领域层使用抽象类和接口，消除对直接具体的类的使用，并提供松耦合。出于演示目的，我们可以简单地使用仍然保留在内存的映射来持久化对象：

```
public interface RestaurantRepository<Restaurant, String> extends
Repository<Restaurant, String> {

    boolean ContainsName(String name);
}

public class InMemRestaurantRepository implements RestaurantRepository
<Restaurant, String> {

    private Map<String, Restaurant> entities;

    public InMemRestaurantRepository() {
        entities = new HashMap();
    }

    @Override
    public boolean ContainsName(String name) {
        return entities.containsKey(name);
    }

    @Override
```

```
    public void add(Restaurant entity) {
        entities.put(entity.getName(), entity);
    }

    @Override
    public void remove(String id) {
        if (entities.containsKey(id)) {
            entities.remove(id);
        }
    }

    @Override
    public void update(Restaurant entity) {
        if (entities.containsKey(entity.getName())) {
            entities.put(entity.getName(), entity);
        }
    }

    @Override
    public boolean contains(String id) {
        throw new UnsupportedOperationException("Not supported yet.");
    //为了修改生成的方法的正文，选择 Tools | Templates。
    }

    @Override
    public Entity get(String id) {
        throw new UnsupportedOperationException("Not supported yet.");
    //为了修改生成的方法的正文，选择Tools | Templates。
    }

    @Override
    public Collection<Restaurant> getAll() {
        return entities.values();
    }

}
```

服务的实现

用前述的方法，可以将领域服务的抽象分割成两个部分：主服务抽象和只读服务的抽象：

```
public abstract class ReadOnlyBaseService<TE, T> {

    private Repository<TE, T> repository;

    ReadOnlyBaseService(Repository<TE, T> repository) {
        this.repository = repository;
    }
    ...
}
```

现在，我们可以使用这个 ReadOnlyBaseService 来创建 BaseService。在这里，我们使用依赖注入模式，通过一个构造函数来映射具有抽象的具体对象：

```
public abstract class BaseService<TE, T> extends
ReadOnlyBaseService<TE, T> {

    private Repository<TE, T> _repository;

    BaseService(Repository<TE, T> repository) {
        super(repository);
        _repository = repository;
    }

    public void add(TE entity) throws Exception {
        _repository.add(entity);
    }

    public Collection<TE> getAll() {
        return _repository.getAll();
    }
}
```

现在，在定义了服务抽象服务后，我们就可以用下列方式实现 RestaurantService：

```
public class RestaurantService extends BaseService<Restaurant, BigInteger> {

    private RestaurantRepository<Restaurant, String> restaurantRepository;

    public RestaurantService(RestaurantRepository repository) {
        super(repository);
        restaurantRepository = repository;
    }

    public void add(Restaurant restaurant) throws Exception {
        if (restaurantRepository.ContainsName(restaurant.getName())) {
            throw new Exception(String.format("There is already a
product with the name - %s", restaurant.getName()));
        }

        if (restaurant.getName() == null || "".equals(restaurant.getName())) {
            throw new Exception("Restaurant name cannot be null or empty
string.");
        }
        super.add(restaurant);
    }
}
```

同样，可以编写其他实体的实现。此代码是一个基本的实现，可以在生产代码中添加各种实现和行为。

小结

本章学习了 DDD 的基本原理，也探讨了多层架构和人们使用 DDD 开发软件可用的不同模式。这时，你可能会意识到领域模型设计对软件的成功是非常重要的。最后，还使用餐馆订座系统来演示了一个领域服务的实现。

在下一章，将学习如何将此设计用于实现示例项目。此示例项目的设计说明是从最后一章摘录的，并将把 DDD 用于生成微服务。这一章不仅包括编码，还包括微服务的不同方面，如构建、单元测试和包装。在下一章末尾，示例微服务项目将可用于部署和使用。

4

实现微服务

本章引导你从我们的示例项目——**在线餐馆订位系统（OTRS）**的设计阶段进入实现阶段。在这里，你将使用上一章所述的相同设计，并增强它，以建立微服务。在这一章的结尾，你不仅将学会实现此设计，还将学到微服务的不同方面——构建、测试和包装。虽然重点是建立和实现 Restaurant 微服务，你可以使用同样的方法来建立和实现 OTRS 中用到的其他微服务。

在这一章，我们将介绍以下主题：

- OTRS 概述
- 开发和实现微服务
- 测试

我们将使用上一章所述的领域驱动设计关键概念。在上一章，你看到了如何将领域驱动设计用于开发使用了核心 Java 的领域模型。现在，我们将从一个示例领域实现转向一个 Spring 框架驱动的实现。你将会利用 Spring Boot 来实现领域驱动设计概念并将它们从核心 Java 转换为基于 Spring 框架的模型。

此外，我们还会使用 Spring Cloud，它提供了一个云就绪的解决方案。Spring Cloud 也使用 Spring Boot，它允许你使用嵌入式应用程序容器，这依靠你的服务内的 Tomcat 或 Jetty，被打包为一个 JAR 文件或 WAR 文件。这个 JAR 作为一个单独的进程执行，也就是将对所有请求提供服务和响应，并指向此服务中定义的端点的一个微服务。

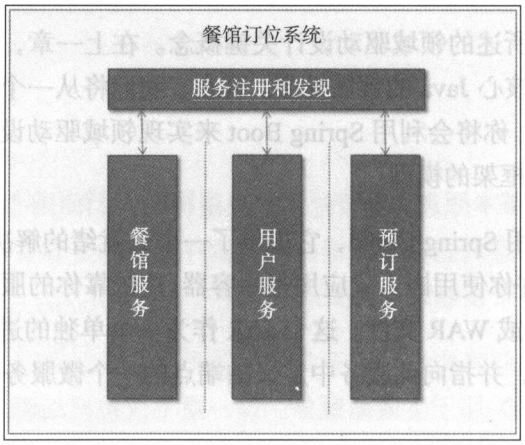
Spring Cloud 也可以方便地与 Netflix Eureka 集成，这是一种服务注册和发现组件。OTRS 将它用于注册和发现微服务。

OTRS 概述

基于微服务的原则，我们需要为每个可以独立执行的功能都建立单独的微服务。观察 OTRS 之后，我们可以轻松地把 OTRS 分为三个主要的微服务——餐馆服务、预订服务和用户服务，还可以在 OTRS 中定义其他的微服务。我们的重点是这三个微服务，要点是让它们独立，包括拥有自己单独的数据库。

我们可以总结出这些服务的功能，如下所示：

- **餐馆服务（Restaurant service）**：此服务提供餐馆资源的如下功能——**创建、读取、更新、删除（CRUD）**操作和基于标准的搜索。它提供餐馆和餐桌之间的关联，餐馆还将提供对 Table 实体的访问。
- **用户服务（User service）**：顾名思义，此服务允许最终用户在 User 实体上执行 CRUD 操作。
- **预订服务（Booking service）**：此服务使用的餐馆服务和用户服务对预订执行 CRUD 操作。它会使用餐馆搜索，与其相关联的餐桌查找，并基于餐桌在指定的时间段的可用性对其进行分配。它会建立 Restaurant/Table 和 User 之间的关系。



不同的微服务、注册和发现

上图显示每个微服务是如何独立地工作的。这是微服务能够分别被开发、增强、维护，而不影响其他服务的原因。这些服务可以分别有自己的分层架构和数据库，并不限于使用相同的技术、框架和语言来开发这些服务。你也可以在任何给定时间点上引入新的微服务。例如，为会计目的，我们可以引入向餐馆服务公开记账的会计服务。同样，分析和报告也都是可以集成和公开的其他服务。

出于演示目的，我们将只实现前面的关系图中所示的三个服务。

开发和实现微服务

我们将使用上一章所述的领域驱动实现和方法来实现利用 Spring Cloud 的微服务。让我们重温以下关键的工件：

- **实体**：这些都是可识别并保持产品或服务的状态不变的对象的类别。这些对象不（**NOT**）由它们的属性定义，而由它们的身份和线程的连续性定义。
  实体拥有诸如身份、连续性的线程，以及不能确定其身份的属性等特征。**值对象**（**VO**）只有属性且没有概念上的身份。最佳的做法是把值对象保持为不可变的对象。在 Spring 框架中，实体是纯 Pojo，因此我们也把它们作为 VO 来使用。
- **服务**：这些都是技术框架中常见的，也用于领域驱动设计的领域层。服务对象不具有内部的状态，它的唯一目的是提供针对领域的行为。服务对象提供不能与特定实体或值对象相关联的行为。服务对象可能会给一个或多个实体或值对象提供一个或多个相关行为。在领域模型中显式定义服务是最佳做法。
- **存储库对象**：存储库对象是领域模型的一部分，它与存储区，如数据库、外部来源等交互，以获取持久化的对象。当存储库收到对某个对象引用的请求时，它返回现有的对象引用。如果存储库中不存在所请求的对象，那么它从存储区获取此对象。

下载示例代码
本书序言中提到下载代码包的详细步骤，请查看。
本书的代码压缩包也驻留在 GitHub 上，位于 https://github.com/PacktPublishing/Mastering-Microservices-with-Java。在我们丰富的书籍和视频目录中也有其他的代码包，请查看 https://github.com/PacktPublishing/。

- 每个 OTRS 微服务 API 都表示一个 REST 式的 web 服务。OTRS API 使用 HTTP 动词，如 GET、POST 等，以及一个 REST 式的端点结构。请求和响应的有效载荷都被格式化为 JSON。如果需要，还可以使用 XML。

餐馆微服务

Restaurant 微服务将以 REST 端点的方式对外部世界公开提供使用。我们会在 Restaurant 微服务示例中找到下列端点，可以根据要求添加任意多的端点：

端点	`GET /v1/restaurants/<Restaurant-Id>`	
参数		
名称	**说明**	
`Restaurant_Id`	路径参数，表示与此 ID 相关联的唯一餐馆	
请求		
属性	**类型**	**说明**
无		
响应		
属性	**类型**	**说明**
`Restaurant`	Restaurant 对象	与给定的 ID 相关联的 Restaurant 对象

端点	`GET /v1/restaurants/`	
参数		
名称	**说明**	
无		
请求		
属性	**类型**	**说明**
`Name`	字符串	表示餐馆的名称或名称的子字符串的查询参数
响应		
属性	**类型**	**说明**
`Restaurants`	Restaurant 对象数组	返回其名称中包含给定名称值的所有餐馆

端点	POST /v1/restaurants/	
参数		
名称	**说明**	
无		
请求		
属性	**类型**	**说明**
Restaurant	Restaurant 对象	餐馆对象的 JSON 表示
响应		
属性	**类型**	**说明**
Restaurant	Restaurant 对象	一个新创建的 Restaurant 对象

同样，我们可以添加许多端点以及它们的实现。出于演示目的，我们将使用 Spring Cloud 来实现前面的端点。

控制器类

Restaurant 控制器使用@RestController 注解来建立这个餐馆服务端点。我们已经在**第 2 章**通览了@RestController 的详细信息。@RestController 是用于资源类的类级别注解，它是@Controller 和@ResponseBody 的组合，它返回领域对象。

API 版本控制

在我们进入下一主题前，我想介绍一下在 REST 端点上使用的 v1 前缀，它表示 API 版本。我也想强调一下 API 版本控制的重要性。因为随着时间的推移，API 会变化，所以版本控制 API 是重要的。随着时间的推移，你的知识和经验更丰富了，从而需要对你的 API 做出更改，API 的更改可能会破坏现有客户端的集成。

因此，有很多种管理 API 版本的办法。其中一种是在路径中使用版本，还有一种方法是使用 HTTP 标头。HTTP 标头可以是表示调用的 API 版本的自定义请求标头或 Accept 标头。请参阅 Packt 出版的 Bhakti Mehta 编写的《*RESTful Java Patterns and Best Practices*》一书，请访问 https://www.packtpub.com/application-development/restful-javapatterns-and-best-practices，以获得更多的信息。

```
@RestController
@RequestMapping("/v1/restaurants")
```

```
public class RestaurantController {

    protected Logger logger = Logger.getLogger(RestaurantController.
class.getName());

    protected RestaurantService restaurantService;

    @Autowired
    public RestaurantController(RestaurantService restaurantService) {
        this.restaurantService = restaurantService;
    }

    /**
     * 获取具有指定名称的餐馆。支持不区分大小的部分
     * 匹配。所以，<code>http://.../restaurants/rest</code> 将会找出
     * 在其名字中有大写或小写 'rest' 的任何餐馆。
     *
     * @param name
     * @返回一个不为空、非空的餐馆集合。
     */
    @RequestMapping(method = RequestMethod.GET)
    public ResponseEntity<Collection<Restaurant>> findByName(@
RequestParam("name") String name) {

logger.info(String.format("restaurant-service findByName() invoked:{}
for {} ", restaurantService.getClass().getName(), name));
        name = name.trim().toLowerCase();
        Collection<Restaurant> restaurants;
        try {
            restaurants = restaurantService.findByName(name);
        } catch (Exception ex) {
            logger.log(Level.WARNING, "Exception raised findByName
REST Call", ex);
            return new ResponseEntity< Collection<
Restaurant>>(HttpStatus.INTERNAL_SERVER_ERROR);
        }
        return restaurants.size() > 0 ? new ResponseEntity<
```

```
Collection< Restaurant>>(restaurants, HttpStatus.OK)
            : new ResponseEntity< Collection<
Restaurant>>(HttpStatus.NO_CONTENT);
    }

    /**
     * 获取具有指定ID的餐馆。
     * <code>http://.../v1/restaurants/{restaurant_id}</code> 将返回
     * 具有指定ID的餐馆。
     *
     * @param retaurant_id
     * @返回一个不为空、非空的餐馆集合。
     */
    @RequestMapping(value = "/{restaurant_id}", method =
RequestMethod.GET)
    public ResponseEntity<Entity> findById(@PathVariable("restaurant_
id") String id) {

      logger.info(String.format("restaurant-service findById()
invoked:{} for {} ", restaurantService.getClass().getName(), id));
       id = id.trim();
       Entity restaurant;
       try {
          restaurant = restaurantService.findById(id);
       } catch (Exception ex) {
          logger.log(Level.SEVERE, "Exception raised findById REST
Call", ex);// findById REST调用引发异常
          return new ResponseEntity<Entity>(HttpStatus.INTERNAL_
SERVER_ERROR);
       }
       return restaurant != null ? new ResponseEntity<Entity>(restaur
ant, HttpStatus.OK)
            : new ResponseEntity<Entity>(HttpStatus.NO_CONTENT);
    }

    /**
```

```
     * 添加具有指定信息的餐馆。
     *
     * @param Restaurant
     * @返回非空的餐馆。
     * @如果根本没有匹配项，则抛出RestaurantNotFoundException。
     */
    @RequestMapping(method = RequestMethod.POST)
    public ResponseEntity<Restaurant> add(@RequestBody RestaurantVO
restaurantVO) {

        logger.info(String.format("restaurant-service add() invoked:
%s for %s", restaurantService.getClass().getName(), restaurantVO.
getName()));

        Restaurant restaurant = new Restaurant(null, null, null);
        BeanUtils.copyProperties(restaurantVO, restaurant);
        try {
            restaurantService.add(restaurant);
        } catch (Exception ex) {
            logger.log(Level.WARNING, "Exception raised add Restaurant
REST Call "+ ex); // add Restaurant REST调用引发异常
            return new ResponseEntity<Restaurant>(HttpStatus.
UNPROCESSABLE_ENTITY);
        }
        return new ResponseEntity<Restaurant>(HttpStatus.CREATED);
    }
}
```

服务类

RestaurantController 使用 RestaurantService。RestaurantServiceis 定义了 CRUD 和一些界面搜索操作，定义如下：

```
public interface RestaurantService {

    public void add(Restaurant restaurant) throws Exception;
```

```
    public void update(Restaurant restaurant) throws Exception;

    public void delete(String id) throws Exception;

    public Entity findById(String restaurantId) throws Exception;

    public Collection<Restaurant> findByName(String name) throws Exception;

    public Collection<Restaurant> findByCriteria(Map<String,
ArrayList<String>> name) throws Exception;
}
```

现在，我们可以实现刚定义的 RestaurantServicewe。它还扩展了在上一章中创建的 BaseService。我们使用@ServiceSpring 注解将其定义为一种服务：

```
@Service("restaurantService")
public class RestaurantServiceImpl extends BaseService<Restaurant,
String>
      implements RestaurantService {

   private RestaurantRepository<Restaurant, String>
restaurantRepository;

   @Autowired
   public RestaurantServiceImpl(RestaurantRepository<Restaurant,
String> restaurantRepository) {
      super(restaurantRepository);
      this.restaurantRepository = restaurantRepository;
   }

   public void add(Restaurant restaurant) throws Exception {
      if (restaurant.getName() == null || "".equals(restaurant.
getName())) {
         throw new Exception("Restaurant name cannot be null or
empty string.");//餐馆名称不能为 null 或空字符串。
      }
```

```
        if (restaurantRepository.containsName(restaurant.getName())) {
            throw new Exception(String.format("There is already a
product with the name - %s", restaurant.getName()));
        }//已经有一种同名产品

        super.add(restaurant);
    }

    @Override
    public Collection<Restaurant> findByName(String name) throws
Exception {
        return restaurantRepository.findByName(name);
    }

    @Override
    public void update(Restaurant restaurant) throws Exception {
        restaurantRepository.update(restaurant);
    }

    @Override
    public void delete(String id) throws Exception {
        restaurantRepository.remove(id);
    }

    @Override
    public Entity findById(String restaurantId) throws Exception {
        return restaurantRepository.get(restaurantId);
    }

    @Override
    public Collection<Restaurant> findByCriteria(Map<String,
ArrayList<String>> name) throws Exception {
        throw new UnsupportedOperationException("Not supported yet.");
//若要更改生成的方法体，选择 Tools | Templates。
    }
}
```

存储库类

RestaurantRepository接口定义了两种新方法:containsName和findByName方法。它还扩展了Repository接口:

```
public interface RestaurantRepository<Restaurant, String> extends
Repository<Restaurant, String> {

    boolean containsName(String name) throws Exception;

    Collection<Restaurant> findByName(String name) throws Exception;
}
```

Repository接口定义了三种方法:and、remove和update。它还扩展了ReadOnlyRepository接口:

```
public interface Repository<TE, T> extends ReadOnlyRepository<TE, T> {

    void add(TE entity);

    void remove(T id);

    void update(TE entity);
}
```

ReadOnlyRepository接口定义包含get和getAll方法,分别返回布尔值、实体和实体的集合。如果你想要只公开只读存储库的抽象,它非常有用:

```
public interface ReadOnlyRepository<TE, T> {

    boolean contains(T id);

    Entity get(T id);

    Collection<TE> getAll();
}
```

Spring框架使用@Repository注解来定义实现存储库的存储库bean。在

RestaurantRepository的例子中，可以看到有一个映射被用来代替实际的数据库实现。这样，只在内存中保存所有的实体。因此，当我们启动服务时，发现在内存中只有两间餐馆。我们可以使用 JPA 来实现数据库持久性，这是用于生产的实现的一般做法：

```
@Repository("restaurantRepository")
public class InMemRestaurantRepository implements RestaurantRepository
<Restaurant, String> {
    private Map<String, Restaurant> entities;

    public InMemRestaurantRepository() {
        entities = new HashMap();
        Restaurant restaurant = new Restaurant("Big-O Restaurant", "1", null);
        entities.put("1", restaurant);
        restaurant = new Restaurant("O Restaurant", "2", null);
        entities.put("2", restaurant);
    }

    @Override
    public boolean containsName(String name) {
        try {
            return this.findByName(name).size() > 0;
        } catch (Exception ex) {
            //异常处理程序
        }
        return false;
    }

    @Override
    public void add(Restaurant entity) {
        entities.put(entity.getId(), entity);
    }

    @Override
    public void remove(String id) {
        if (entities.containsKey(id)) {
            entities.remove(id);
        }
```

```
    }

    @Override
    public void update(Restaurant entity) {
        if (entities.containsKey(entity.getId())) {
            entities.put(entity.getId(), entity);
        }
    }

    @Override
    public Collection<Restaurant> findByName(String name) throws
Exception {
        Collection<Restaurant> restaurants = new ArrayList();
        int noOfChars = name.length();
        entities.forEach((k, v) -> {
            if (v.getName().toLowerCase().contains(name.subSequence(0,
noOfChars))) {
                restaurants.add(v);
            }
        });
        return restaurants;
    }

    @Override
    public boolean contains(String id) {
        throw new UnsupportedOperationException("Not supported yet.");
//若要更改生成的方法体，选择 Tools | Templates。
    }
    @Override
    public Entity get(String id) {
        return entities.get(id);
    }

    @Override
    public Collection<Restaurant> getAll() {
        return entities.values();
    }
}
```

实体类

Restaurant 实体扩展了 BaseEntity，它的定义如下：

```
public class Restaurant extends BaseEntity<String> {

    private List<Table> tables = new ArrayList<>();

    public Restaurant(String name, String id, List<Table> tables) {
        super(id, name);
        this.tables = tables;
    }

    public void setTables(List<Table> tables) {
        this.tables = tables;
    }

    public List<Table> getTables() {
        return tables;
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append(String.format("id: {}, name: {}, capacity: {}",
this.getId(), this.getName(), this.getCapacity()));
        return sb.toString();
    }
}
```

因为我们为实体的定义使用 POJO 类，所以在很多情况下不需要创建一个值对象。这个思路是，不应在实体之间保持对象的状态。

Table 实体扩展了 BaseEntity，它的定义如下：

```
public class Table extends BaseEntity<BigInteger> {
    private int capacity;
```

```
    public Table(String name, BigInteger id, int capacity) {
        super(id, name);
        this.capacity = capacity;
    }
    public void setCapacity(int capacity) {
        this.capacity = capacity;
    }
    public int getCapacity() {
        return capacity;
    }
    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append(String.format("id: {}, name: {}", this.getId(),
this.getName()));
        sb.append(String.format("Tables: {}" +
Arrays.asList(this.getTables())));
        return sb.toString();
    }
}
```

Entity 抽象类定义如下：

```
public abstract class Entity<T> {

    T id;
    String name;

    public T getId() {
        return id;
    }

    public void setId(T id) {
        this.id = id;
    }

    public String getName() {
        return name;
```

```
    }

    public void setName(String name) {
        this.name = name;
    }

}
```

BaseEntity 抽象类的定义如下，它扩展了 Entity 抽象类：

```
public abstract class BaseEntity<T> extends Entity<T> {

    private T id;
    private boolean isModified;
    private String name;

    public BaseEntity(T id, String name) {
        this.id = id;
        this.name = name;
    }

    public T getId() {
        return id;
    }

    public void setId(T id) {
        this.id = id;
    }

    public boolean isIsModified() {
        return isModified;
    }

    public void setIsModified(boolean isModified) {
        this.isModified = isModified;
    }

    public String getName() {
```

```
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

}
```

预订和用户服务

我们可以使用 `RestaurantService` 实现来开发预订（Booking）和用户（User）服务。User 服务能提供与 CRUD 操作有关的用户资源的端点。Booking 服务可以提供与 CRUD 操作和餐桌位子的可用性相关的预订资源的端点。可以在 Packt 网站上找到这些服务的示例代码。

注册和发现服务（Eureka 服务）

Spring Cloud 提供了对 *Netflix Eureka* 最先进的支持，这是一种服务注册和发现工具。所有由你执行的服务都被 Eureka 服务列出和发现，这是从你的服务项目内的 Eureka 客户端 Spring 配置中读取的。

它需要如下所示的 Spring Cloud 依赖项和在 `pom.xml` 中的启动类，其中包含 `@EnableEurekaApplication` 注解：

Maven 依赖项：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka-server</artifactId>
</dependency>
```

启动类：

启动类 **App** 将只使用`@EnableEurekaApplication` 类注解来无缝地运行 Eureka 服务：

```
package com.packtpub.mmj.eureka.service;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.
EnableEurekaServer;

@SpringBootApplication
@EnableEurekaServer
public class App {

    public static void main(String[] args) {
        SpringApplication.run(App.class, args);
    }
}
```

使用 pom.xml 项目中的 <properties>标记下的<start-class>com.packtpub.mmj.eureka.service.App</start-class>。

Spring 的配置：

Eureka 服务也需要下列 Spring 配置信息用于 Eureka 服务器配置(src/main/resources/application.yml):

```
server:
  port: ${vcap.application.port:8761}   # HTTP 端口

eureka:
  instance:
    hostname: localhost
  client:
    registerWithEureka: false
    fetchRegistry: false
  server:
    waitTimeInMsWhenSyncEmpty: 0
```

类似于 Eureka 服务器，每个 OTRS 服务还应包含 Eureka 客户端配置，以便可以建立 Eureka 服务器和客户端之间的连接。没有这一点，是不可能注册和发现服务的。

Eureka 客户端：你的服务可以使用以下的 spring 配置来配置 Eureka 服务器：

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

执行

为了查看我们的代码如何工作，需要首先生成它，然后执行它。我们将使用 Maven clean package 来建立服务 JAR。

现在若要执行这些服务 JAR 文件，只需从服务主目录执行下面的命令：

```
java -jar target/<service_jar_file>
```

例如：

```
java -jar target/restaurant-service.jar
java -jar target/eureka-service.jar
```

测试

若要启用测试，需要在 pom.xml 中添加以下依赖项：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
</dependency>
```

为了测试 RestaurantController，已添加了下列文件：

- RestaurantControllerIntegrationTests，使用@SpringApplicationConfiguration 注解来选择 Spring Boot 使用的相同配置：

```
@RunWith(SpringJUnit4ClassRunner.class)
```

```
@SpringApplicationConfiguration(classes = RestaurantApp.class)
public class RestaurantControllerIntegrationTests extends
      AbstractRestaurantControllerTests {
}
```

- 一个抽象类来编写我们的测试：

```
public abstract class AbstractRestaurantControllerTests {

    protected static final String RESTAURANT = "1";
    protected static final String RESTAURANT_NAME = "Big-O Restaurant";

    @Autowired
    RestaurantController restaurantController;

    @Test
    public void validResturantById() {
        Logger.getGlobal().info("开始validResturantById 测试");
        ResponseEntity<Entity> restaurant = restaurantController.
findById(RESTAURANT);

        Assert.assertEquals(HttpStatus.OK, restaurant.
getStatusCode());
        Assert.assertTrue(restaurant.hasBody());
        Assert.assertNotNull(restaurant.getBody());
        Assert.assertEquals(RESTAURANT, restaurant.getBody().
getId());
        Assert.assertEquals(RESTAURANT_NAME, restaurant.getBody().
getName());
        Logger.getGlobal().info("结束validResturantById 测试");
    }

    @Test
    public void validResturantByName() {
        Logger.getGlobal().info("开始validResturantByName 测试");
        ResponseEntity<Collection<Restaurant>> restaurants =
restaurantController.findByName(RESTAURANT_NAME);
```

```
        Logger.getGlobal().info("正在 validAccount 测试");

        Assert.assertEquals(HttpStatus.OK, restaurants.
getStatusCode());
        Assert.assertTrue(restaurants.hasBody());
        Assert.assertNotNull(restaurants.getBody());
        Assert.assertFalse(restaurants.getBody().isEmpty());
        Restaurant restaurant = (Restaurant) restaurants.
getBody().toArray()[0];
        Assert.assertEquals(RESTAURANT, restaurant.getId());
        Assert.assertEquals(RESTAURANT_NAME, restaurant.
getName());
        Logger.getGlobal().info("结束validResturantByName 测试");
    }

    @Test
    public void validAdd() {
        Logger.getGlobal().info("开始validAdd 测试");
        RestaurantVO restaurant = new RestaurantVO();
        restaurant.setId("999");
        restaurant.setName("测试Restaurant");

        ResponseEntity<Restaurant> restaurants =
restaurantController.add(restaurant);
        Assert.assertEquals(HttpStatus.CREATED, restaurants.
getStatusCode());
        Logger.getGlobal().info("结束validAdd 测试");
    }
}
```

- 最后，RestaurantControllerTests 扩展了以前创建的抽象类，也创建了 RestaurantService 和 RestaurantRepository 实现：

```
public class RestaurantControllerTests extends
AbstractRestaurantControllerTests {

    protected static final Restaurant restaurantStaticInstance =
```

```
new Restaurant(RESTAURANT,
        RESTAURANT_NAME, null);

   protected static class TestRestaurantRepository implements Res
taurantRepository<Restaurant, String> {

      private Map<String, Restaurant> entities;

      public TestRestaurantRepository() {
         entities = new HashMap();
         Restaurant restaurant = new Restaurant("Big-O Restaurant", "1",
null);
         entities.put("1", restaurant);
         restaurant = new Restaurant("O Restaurant", "2", null);
         entities.put("2", restaurant);
      }

      @Override
      public boolean containsName(String name) {
         try {
            return this.findByName(name).size() > 0;
         } catch (Exception ex) {
            //异常处理程序
         }
         return false;
      }

      @Override
      public void add(Restaurant entity) {
         entities.put(entity.getId(), entity);
      }

      @Override
      public void remove(String id) {
         if (entities.containsKey(id)) {
            entities.remove(id);
         }
```

```
    }

    @Override
    public void update(Restaurant entity) {
        if (entities.containsKey(entity.getId())) {
            entities.put(entity.getId(), entity);
        }
    }
    @Override
    public Collection<Restaurant> findByName(String name)
throws Exception {
        Collection<Restaurant> restaurants = new ArrayList();
        int noOfChars = name.length();
        entities.forEach((k, v) -> {
            if (v.getName().toLowerCase().contains(name.
subSequence(0, noOfChars))) {
                restaurants.add(v);
            }
        });
        return restaurants;
    }

    @Override
    public boolean contains(String id) {
        throw new UnsupportedOperationException("Not supported
yet."); //若要更改生成的方法体，选择 Tools | Templates。
    }

    @Override
    public Entity get(String id) {
        return entities.get(id);
    }

    @Override
    public Collection<Restaurant> getAll() {
        return entities.values();
    }
```

```
    }

    protected TestRestaurantRepository testRestaurantRepository =
new TestRestaurantRepository();
    protected RestaurantService restaurantService = new Restaurant
ServiceImpl(testRestaurantRepository);

    @Before
    public void setup() {
        restaurantController = new RestaurantController(restaurant
Service);
    }
}
```

参考资料

- *RESTful Java Patterns and Best Practices*（REST 式的 Java 模式和最佳实践），Bhakti Mehta 著，Packt 出版社：`https://www.packtpub.com/application-development/restful-javapatterns-and-best-practices`
- *Spring Cloud*：`http://cloud.spring.io/`
- *Netflix Eureka*：`https://github.com/netflix/eureka`

小结

本章我们学习了如何在微服务中使用领域驱动设计模型。运行演示应用程序后，我们可以看到如何独立地开发、部署，并测试每个微服务。可以很方便地使用 Spring Cloud 来创建微服务。同时，我们也探讨如何使用 Eureka 注册和发现 Spring Cloud 组件。

在下一章中，我们将学习在诸如 Docker 的容器中部署微服务，还将了解使用其他 Java 客户端和其他工具来测试微服务。

5

部署和测试

这一章将会解释如何用不同的形式，包括独立部署和使用诸如 Docker 的容器来部署微服务，还将演示如何用 Docker 把我们的示例项目部署到云服务如 AWS 上。在实现 Docker 之前，我们将首先探索微服务的其他相关因素，如负载均衡和边缘服务器，你也将了解使用不同的 REST 客户端，如 RestTemplate、Netflix Feign 等来测试微服务。

在这一章，我们将介绍以下主题：

- 使用 Netflix OSS 的微服务架构概述
- 微服务的负载均衡
- 边缘服务器
- 断路器和监控
- 使用容器部署微服务
- 使用 Docker 容器对微服务进行集成测试

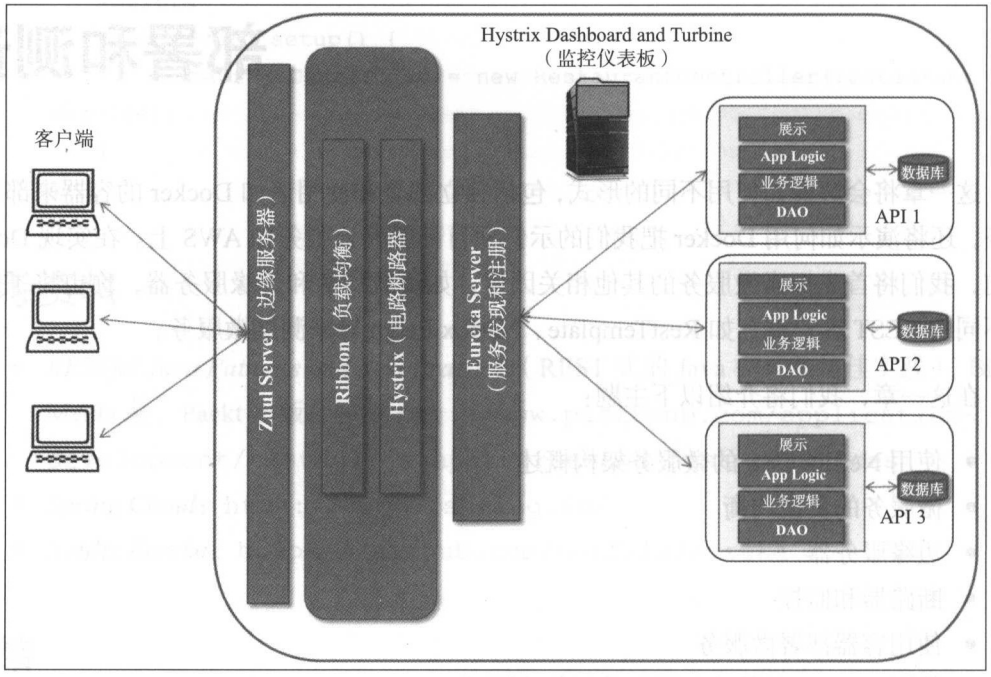
使用 Netflix OSS 的微服务架构概述

Netflix 是微服务架构中的先锋。它们是第一个成功在很大程度上实现微服务架构的组织，它们也有助于提高其受欢迎程度，并通过 **Netflix 开放源码软件中心（Open Source Software Center，OSS）**开源它们的大多数微服务工具，为推广微服务做出了巨大的贡献。

按照 Netflix 博客的描述，当 Netflix 开发自己的平台时，它们使用 Apache Cassandra 的数据存储，这是来自 Apache 的一种开源工具。它们开始通过修复和优化扩展为 Cassandra

做贡献。这使得 Netflix 看到用开放源码软件中心这个名称来发布 Netflix 项目的好处。

Spring 借此机会，把许多 Netflix 开放源码软件项目，如 Zuul、Ribbon、Hystrix、Eureka Server 和 Turbine，都集成到 Spring Cloud 中。这是 Spring Cloud 能够为开发可用于生产的微服务提供现成平台的原因之一。现在，让我们看看几个重要的 Netflix 工具以及它们是如何适应微服务架构的。



微服务架构关系图

可以在上图中看到，对于每个微服务的做法，我们都有与它关联的 Netflix 工具。我们可以通过下面的映射来理解它。除了 Eureka 已在上一章中详细阐述了，其他工具的详细信息都在本章各节进行探讨。

- **边缘服务器**：我们使用 Netflix Zuul Server 作为边缘服务器。
- **负载均衡**：Netflix Ribbon 用于负载均衡。
- **电路断路器**：Netflix Hystrix 用作电路断路器和帮助保持系统运行。
- **服务发现和注册**：Netflix Eureka 服务器用于服务发现和注册。

- **监控仪表板**：Hystrix 仪表板与 Netflix Turbine 结合用于微服务监控。它提供了一个仪表板来检查运行中的微服务的健康状况。

负载均衡

如果要以速度快、容量利用率最大化的方式为请求提供服务，就需要实现负载均衡，这样可以确保没有服务器的请求超负荷。如果一台服务器出现故障，负载均衡器也会将请求重定向到其余的主机服务器上。在微服务架构中，微服务可以为内部或外部的请求提供服务，在此基础上，我们可以有两种类型的负载均衡——客户端负载均衡和服务器端负载均衡。

客户端的负载均衡

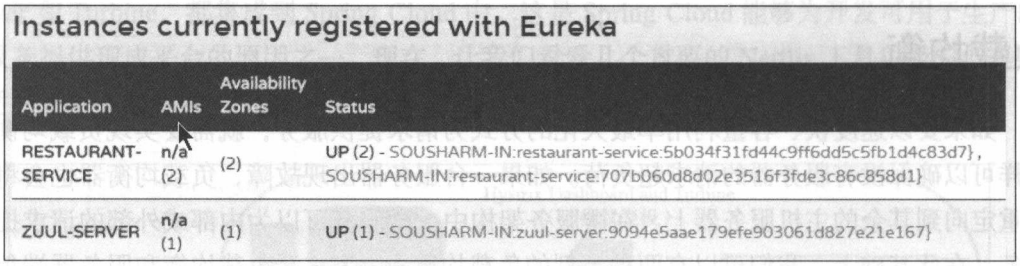
微服务需要使用进程间通信，以便服务可以互相交流。Spring Cloud 使用 Netflix Ribbon 承担负载均衡器这个至关重要的角色，并可以处理 HTTP 和 TCP 客户端的负载。Ribbon 可用于云平台，并提供内置的故障恢复能力。Ribbon 还允许使用多个和可插拔的负载均衡规则，它集成了客户端和负载均衡器。

在上一章，我们添加了 Eureka 服务器。在 Spring Cloud 中，Ribbon 在默认情况下与 Eureka 服务器集成。这种集成提供了以下功能：

- 在使用 Eureka 服务器时，不需要对要发现的远程服务器 URL 进行硬编码。这是一个突出的优势，尽管你仍然可以根据需要在 `application.yml` 中使用已配置的服务器列表（*listOfServers*）。
- 服务器列表由 Eureka 服务器来填充。Eureka 服务器用 `DiscoveryEnabledNIWSServerList` 重写 `ribbonServerList`。
- 找出服务器是否在运行的请求被委托给 Eureka。`DiscoveryEnabledNIWSServerList` 接口用于替代 Ribbon 的 `IPing`。

Spring Cloud 中有使用 Ribbon 的不同客户端，如 **RestTemplateor** 或 **FeignClient**。这些客户端允许微服务间相互通信。使用 Eureka 服务器时，客户端使用实例 ID 代替主机名和端口用于 HTTP 向服务实例发出调用。客户端把服务 ID 传给 Ribbon，Ribbon 然后使用负载均衡器获取 Eureka 服务器实例。

如果在 Eureka 中有可用的服务的多个实例，如下面的屏幕截图所示，Ribbon 基于负载均衡算法，只为请求获取一个实例：



多个服务注册——餐馆服务

我们可以使用 DiscoveryClient 来查找 Eureka 服务器中可用的服务的所有实例，如下面的代码所示。DiscoveryClientSample 类的 getLocalServiceInstance() 方法返回 Eureka 服务器中所有可用的本地服务实例。

DiscoveryClient 示例：

```
@Component
class DiscoveryClientSample implements CommandLineRunner {

    @Autowired
    private DiscoveryClient;

    @Override
    public void run(String... strings) throws Exception {
   //打印发现客户端的描述
        System.out.println(discoveryClient.description());
   //获得餐馆服务实例并打印它的信息
        discoveryClient.getInstances("restaurant-service").
forEach((ServiceInstance serviceInstance) -> {
            System.out.println(new StringBuilder("Instance -->
").append(serviceInstance.getServiceId())
                    .append("\nServer: ").append(serviceInstance.
getHost()).append(":").append(serviceInstance.getPort())
                    .append("\nURI: ").append(serviceInstance.
getUri()).append("\n\n\n"));
```

```
        });
    }
}
```

在执行时，此代码将打印以下信息。它显示了餐馆服务的两个实例：

```
Spring Cloud Eureka Discovery Client
Instance: RESTAURANT-SERVICE
Server: SOUSHARM-IN:3402
URI: http://SOUSHARM-IN:3402
Instance --> RESTAURANT-SERVICE
Server: SOUSHARM-IN:3368
URI: http://SOUSHARM-IN:3368
```

下面的示例展示如何使用这些客户端。可以看到，在两个客户端中，都用服务名 restaurant-service 来替代服务的主机名和端口。这些客户端调用/v1/restaurants 来获取在名称查询参数中所包含的名字的餐馆列表：

Rest 模板示例：

```
@Override
public void run(String... strings) throws Exception {
ResponseEntity<Collection<Restaurant>> exchange
= this.restTemplate.exchange(
"http://restaurant-service/v1/restaurants?name=o",
                HttpMethod.GET,
                null,
                new ParameterizedTypeReference<Collection<Restaurant>>() {
               },
                ( "restaurants");
exchange.getBody().forEach((Restaurant restaurant) -> {
System.out.println(new StringBuilder("\n\n\n[ ").append(restaurant.
getId()).append(" ").append(restaurant.getName()).append("]"));
});
}
```

FeignClient 示例：

```
@Component
```

```
class FeignSample implements CommandLineRunner {

    @Autowired
    private RestaurantClient restaurantClient;

    @Override
    public void run(String... strings) throws Exception {
        this.restaurantClient.getRestaurants("o").forEach((Restaurant
restaurant) -> {
            System.out.println(restaurant);
        });
    }
}

@FeignClient("restaurant-service")
interface RestaurantClient {

    @RequestMapping(method = RequestMethod.GET, value = "/v1/restaurants")
    Collection<Restaurant> getRestaurants(@RequestParam("name") String
name);
}
```

上述所有示例将都打印以下输出：

```
[ 1 Big-O Restaurant]
[ 2 O Restaurant]
```

服务器端的负载均衡

在实现客户端的负载均衡后，定义服务器端的负载均衡是重要的。此外，从微服务架构的角度来看，定义我们的 OTRS 应用程序的路由机制是很重要的。例如，/可能被映射到我们的 UI 应用程序，/restaurantapi 被映射到餐馆服务，而/userapi 被映射到用户服务。

我们将使用 Netflix Zuul Server 作为我们的边缘服务器。Zuul 是一个基于 JVM 的路由器和服务器端的负载均衡器。Zuul 支持用任何 JVM 语言来编写规则和过滤条件并具有内置的 Java 和 Groovy 的支持。

外部世界（UI 和其他客户端）调用边缘服务器，它使用在 application.yml 中定义的路由，调用内部服务并提供响应。如果你认为它充当代理服务器、承担内部网络的网关，并为定义和配置的路由调用内部服务，你的猜测是正确的。

通常情况下，推荐为所有请求提供一个单独的边缘服务器。然而，少数几家公司为每个客户端使用单独的边缘服务器，以便扩展。例如，Netflix 为每种设备类型使用专用的边缘服务器。

在下一章，当我们配置并实现微服务的安全性时，也将使用边缘服务器。

在 Spring Cloud 中配置和使用边缘服务器非常简单，只需要采取以下步骤：

1. 在 pom.xml 中定义 Zuul 服务器依赖项：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zuul</artifactId>
</dependency>
```

2. 在应用程序类中使用 @EnableZuulProxy 注解。它还会在内部使用 @EnableDiscoveryClient：因此它也会自动注册到 Eureka 服务器。可以在最后一个图形：多个服务注册——餐馆服务中找到已注册的 Zuul 服务器。

3. 在 application.yml 中更新 Zuul 配置，如下所示：

   — zuul:ignoredServices：这跳过服务的自动添加。我们可以在这里定义服务 ID 模式。*表示我们忽略了所有的服务。在以下示例中，除了 restaurant-service 外的所有服务都将被忽略。

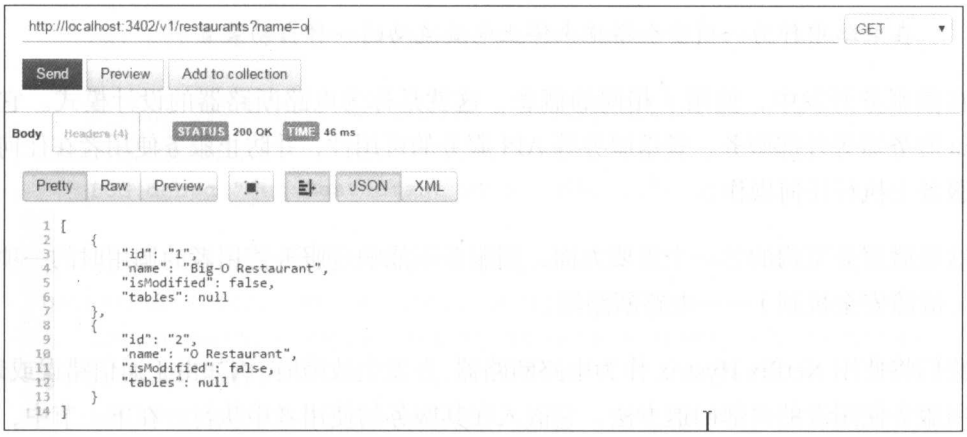
   — Zuul.routes:这包含定义URI模式的path属性。在这里，/restaurantapi 使用 serviceId 被映射到餐馆服务。serviceId 表示在 Eureka 服务器中的服务。如果不使用 Eureka 服务器，可以使用服务的 URL 来替换。我们也用 stripPrefix 属性来剥离前缀(/restaurantapi)，由此导致在调用此服务时，/restaurantapi/v1/estaurants/1 调用将转换为/v1/restaurants/1：

application.yml

```
info:
```

```
    component: Zuul Server
# Spring的属性
spring:
  application:
    name: zuul-server  #服务注册在这个名字下

endpoints:
    restart:
        enabled: true
    shutdown:
        enabled: true
    health:
        sensitive: false

zuul:
    ignoredServices: "*"
    routes:
        restaurantapi:
            path: / restaurantapi/**
            serviceId: restaurant-service
            stripPrefix: true

server:
    port: 8765

# 发现服务器访问
eureka:
  instance:
    leaseRenewalIntervalInSeconds: 3
    metadataMap:
      instanceId: ${vcap.application.instance_id:${spring.
application.name}:${spring.application.instance_id:${random.
value}}}
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
    fetchRegistry: false
```

让我们看看工作中的边缘服务器。首先，我们调用部署在端口 3402 上的餐馆服务，如下图所示。



直接的餐馆服务调用

然后，我们会使用部署在端口 8765 上的边缘服务器调用相同的服务。可以看到，调用/v1/restaurants?name=o 用到了/restaurantapi 前缀，并且它给出了相同的结果。

使用边缘服务器的餐馆服务调用

电路断路器与监控

总体而言，电路断路器是：

> 在电路中作为一项安全措施来停止电流流动的一种自动装置。

在微服务开发中，使用了相同的概念，这就是称为电路断路器的设计模式。它跟踪 Eureka 服务器等外部服务、餐馆服务等 API 服务的可用性，并防止服务使用者在任何不可用的服务上执行任何操作。

这是微服务架构的另一个重要方面，当服务不能响应服务使用者的调用时的一项安全措施（故障安全机制）——电路断路器。

我们将使用 Netflix Hystrix 作为电路断路器。在发生故障时（例如由于通信错误或超时），它调用服务使用者的内部回退方法，它嵌入在其服务的使用者中执行。在下一节中，你会看到实现此功能的代码。

Hystrix 打开电路并在服务多次无法响应时快速断开，直到此服务再次可用时再打开。你一定会想，如果 Hystrix 打开电路，那么它如何知道此服务是可用的呢？它也例外地允许一些请求来调用此服务。

使用 Hystrix 的回退方法

为了实现回退方法，需要采取三个步骤：

1. **启用断路器**：使用其他服务的微服务的主类应使用`@EnableCircuitBreaker`注解。例如，如果一个用户服务想要得到用户已预订了餐桌的餐馆的详细信息：

```
@SpringBootApplication
@EnableCircuitBreaker
@ComponentScan({"com.packtpub.mmj.user.service", "com.packtpub.
mmj.common"})
public class UsersApp {
```

2. **配置回退方法**：使用`@HystrixCommand`注解来配置`fallbackMethod`：

```
@HystrixCommand(fallbackMethod = "defaultRestaurant")
public ResponseEntity<Restaurant> getRestaurantById(int
```

```
restaurantId) {

    LOG.debug("Get Restaurant By Id with Hystrix protection");

    URI uri = util.getServiceUrl("restaurant-service");

    String url = uri.toString() + "/v1/restaurants/" +
restaurantId;
    LOG.debug("Get Restaurant By Id URL: {}", url);

    ResponseEntity<Restaurant> response = restTemplate.
getForEntity(url, Restaurant.class);
    LOG.debug("Get Restaurant By Id http-status: {}", response.
getStatusCode());
    LOG.debug("GET Restaurant body: {}", response.getBody());

    Restaurant restaurant = response.getBody();
    LOG.debug("Restaurant ID: {}", restaurant.getId());

    return serviceHelper.createOkResponse(restaurant);
}
```

3. **定义回退方法**：处理故障和安全执行步骤的一个方法：

```
public ResponseEntity<Restaurant> defaultRestaurant(int
restaurantId) {
    LOG.warn("餐馆服务回退方法正在使用");
    return serviceHelper.createResponse(null, HttpStatus.BAD_
GATEWAY);
}
```

这些步骤足以对服务调用进行故障保护，并向服务使用者返回更为合适的响应。

监控

Hystrix 提供 web 用户界面的仪表板，它提供漂亮的电路断路器图形界面。

默认Hystrix仪表板

Netflix Turbine 是一个 web 应用程序，它连接到集群中 Hystrix 应用程序的实例并会实时（每隔 0.5 秒更新）聚合信息。Turbine 使用一个称为 Turbine 流的流来提供信息。

如果把 Hystrix 与 Netflix Turbine 结合使用，那么可以在 Hystrix 的仪表板上从所有 Eureka 服务器获得信息。这使你获得与电路断路器有关的所有信息的全貌。

要结合使用 Turbine 和 Hystrix，只需在之前显示的第一个文本框中键入 Turbine URL `http://localhost:8989/turbine.stream`(在 `application.yml` 中为 Turbine 服务器配置的端口是 `8989`)，并点击 **Monitory Stream**（**监控流**）。

Netflix Hystrix 和 Turbine 使用 RabbitMQ 这个开源消息队列软件。RabbitMQ 使用**高级消息队列协议**（**Advance Messaging Queue Protocol，AMQP**）工作。可以在这个软件中定义队列，应用程序可以借助它建立连接，并通过它传递消息。一条消息可以包含任何类型的信息。可以在 RabbitMQ 队列中存储消息，直到接收应用程序连接到它并接收此消息为止（将消息从队列中取走）。

Hystrix 使用 RabbitMQ 来发送馈送给 Turbine 的度量数据。

在配置 Hystrix 和 Turbine 前，请在你的平台上安装 RabbitMQ 应用程序。Hystrix 和 Turbine 使用 RabbitMQ 在两者之间进行通信。

设置 Hystrix 仪表板

我们将为 Hystrix 服务器添加 Maven 新依赖项，`dashboard-server`。像其他环境一样，在 `Spring Cloud` 中配置和使用 Hystrix 仪表板是很简单的，只需要按照下列步骤操作：

1. 在 pom.xml 中定义 Hystrix 仪表板依赖项：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter- hystrix-dashboard</
artifactId>
</dependency>
```

2. 使用主 Java 类中的`@EnableHystrixDashboard` 注解完成了全部工作。我们也将使用`@Controller` 把来自根的请求转发到 Hystrix，如下所示：

```
@SpringBootApplication
@Controller
@EnableHystrixDashboard
public class DashboardApp extends SpringBootServletInitializer {

    @RequestMapping("/")
    public String home() {
        return "forward:/hystrix";
    }

    @Override
    protected SpringApplicationBuilder configure(SpringApplication
Builder application) {
        return application.sources(DashboardApp.class).web(true);
```

```
    }

    public static void main(String[] args) {
        SpringApplication.run(DashboardApp.class, args);
    }
}
```

3. 在 application.yml 中更新 Dashboard 应用程序配置，如下所示：

```
application.yml
# Hystrix 仪表板属性
spring:
    application:
        name: dashboard-server

endpoints:
    restart:
        enabled: true
    shutdown:
        enabled: true

server:
    port: 7979

eureka:
    instance:
        leaseRenewalIntervalInSeconds: 3
        metadataMap:
            instanceId: ${vcap.application.instance_id:${spring.
application.name}:${spring.application.instance_id:${random.
value}}}

    client:
        # 来自org.springframework.cloud的默认值。
netflix.eurek.EurekaClientConfigBean
        registryFetchIntervalSeconds: 5
        instanceInfoReplicationIntervalSeconds: 5
```

```
        initialInstanceInfoReplicationIntervalSeconds: 5
        serviceUrl:
            defaultZone: http://localhost:8761/eureka/
        fetchRegistry: false

logging:
    level:
        ROOT: WARN
        org.springframework.web: WARN
```

设置 Turbine

我们会为 Turbine 再创建一个 Maven 依赖项。Hystrix 仪表板应用程序运行时，它会像前面的默认 *Hystrix 仪表板*截图那样显示。

现在，我们将使用以下步骤配置 Turbine 服务器：

1. 在 pom.xml 中定义 Turbine 服务器依赖项：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-turbine-amqp</artifactId>
</dependency>
```

2. 在应用程序类中使用@EnableTurbineAmqp 注解，如下所示。我们也定义了一个 bean，它会返回 RabbitMQ 连接工厂：

```
@SpringBootApplication
@EnableTurbineAmqp
@EnableDiscoveryClient
public class TurbineApp {

    private static final Logger LOG = LoggerFactory.
getLogger(TurbineApp.class);

    @Value("${app.rabbitmq.host:localhost}")
    String rabbitMQHost;
```

```
    @Bean
    public ConnectionFactory connectionFactory() {
        LOG.info("Creating RabbitMQHost ConnectionFactory for
host: {}", rabbitMQHost);
        CachingConnectionFactory cachingConnectionFactory = new
CachingConnectionFactory(rabbitMQHost);
        return cachingConnectionFactory;
    }

    public static void main(String[] args) {
        SpringApplication.run(TurbineApp.class, args);
    }
}
```

3. 在 application.yml 中更新 Turbine 配置，如下所示：

```
server:port: Turbine HTTP使用的主端口
management:port: Turbine执行器端点的端口
application.yml
spring:
    application:
        name: turbine-server

server:
    port: 8989

management:
    port: 8990

PREFIX:

endpoints:
    restart:
        enabled: true
    shutdown:
        enabled: true
```

```
eureka:
    instance:
        leaseRenewalIntervalInSeconds: 10
    client:
        registryFetchIntervalSeconds: 5
        instanceInfoReplicationIntervalSeconds: 5
        initialInstanceInfoReplicationIntervalSeconds: 5
        serviceUrl:
            defaultZone: http://localhost:8761/eureka/

logging:
    level:
        root: WARN
        com.netflix.discovery: 'OFF'
```

请注意前面的步骤始终使用默认配置创建各自的服务器。如果需要，你可以用特定的设置重写默认配置。

使用容器部署微服务

阅读**第 1 章**后，你可能已经了解了有关 Docker 的要点。

Docker 容器提供轻量级的运行时环境，它由虚拟机的核心功能和一个称为 Docker 映像的隔离的操作系统服务组成。Docker 使得包装和执行微服务更加容易、顺畅。每个操作系统可以有多个 Docker，并且每个 Docker 都可以运行多个应用程序。

安装和配置

如果你不使用 Linux 操作系统，那么 Docker 需要一台虚拟化的服务器。可以安装 VirtualBox 或类似的工具，如 Docker 工具箱来使它为你工作。Docker 安装页面提供了有关它的详细信息，并告诉你如何去做。所以，可在 Docker 的网站上阅读 Docker 安装指南。

可以按照在 `https://docs.docker.com/engine/installation/`中给出的说明，基于你的平台安装 Docker。

DockerToolbox 1.9.1f 是写作时可用的最新版本。这是我们使用的版本。

具有 4 GB 内存的 Docker 机器

默认情况下创建的机器都有 2GB 的内存。我们将重新创建具有 4GB 内存的 Docker 机器：

```
docker-machine rm default
docker-machine create -d virtualbox --virtualbox-memory 4096 default
```

使用 Maven 构建 Docker 映像

有各种 Docker maven 插件可以用来构建映像：

- `https://github.com/rhuss/docker-maven-plugin`
- `https://github.com/alexec/docker-maven-plugin`
- `https://github.com/spotify/docker-maven-plugin`

可以根据你的选择使用任何一种。我发现由`@rhuss` 提供的 Docker Maven 插件最适合我们。它会定期更新，并且与其他的插件相比，有很多额外的功能。

在开始讨论 `docker-maven-plugin` 的配置前，我们需要介绍 `application.yml` 中的 Docker Spring 配置文件。在为各种平台构建服务时，它将使我们的工作更容易。我们需要配置以下四个属性：

- 我们会使用确定为 Docker 的 Spring 配置文件。
- 嵌入式 Tomcat 的端口之间不会有任何冲突，因为服务将在自己各自的容器内执行。我们现在可以使用端口 `8080`。
- 我们偏爱在 Eureka 中使用我们的服务的 IP 地址。因此，Eureka 实例属性 `preferIpAddress` 将被设置为 `true`。
- 最后，我们将在 `serviceUrl:defaultZone` 中使用 Eureka 服务器主机名。

为了把 Spring 配置文件添加到项目中，需要在 `application.yml` 中现有的内容后添加以下行：

```
---
#用于在Docker容器中部署
spring:
  profiles: docker

server:
  port: 8080

eureka:
  instance:
    preferIpAddress: true
  client:
    serviceUrl:
      defaultZone: http://eureka:8761/eureka/
```

在生成 Docker 容器 JAR 时，我们还将在 pom.xml 中添加以下代码来激活 Spring 配置文件 Docker（这将使用先前定义的属性创建 JAR，例如端口：8080）。

```
<profiles>
    <profile>
        <id>docker</id>
        <properties>
            <spring.profiles.active>docker</spring.profiles.active>
        </properties>
    </profile>
</profiles>
```

我们构建服务时只需要使用 Maven docker 配置文件，如下所示：

```
mvn -P docker clean package
```

上面的命令将生成包含 Tomcat 的 8080 端口的 service JAR，并将在 Eureka 服务器上用主机名 Eureka 注册。

现在，让我们配置 docker-maven-plugin 生成包含餐馆微服务的映像。这个插件必须首先创建一个 Dockerfile。Dockerfile 是在两个地方配置的——在 pom.xml 和 docker-assembly.xml 中。我们将在 pom.xml 中使用如下插件配置：

```
<properties>
<!—对于 Docker hub 保留空白; 对于本地Docker Registry使用 "localhost:5000/" -->
  <docker.registry.name>localhost:5000/</docker.registry.name>
  <docker.repository.name>${docker.registry.name}sourabhh /${project.
artifactId}</docker.repository.name>
</properties>
...
<plugin>
  <groupId>org.jolokia</groupId>
  <artifactId>docker-maven-plugin</artifactId>
  <version>0.13.7</version>
  <configuration>
    <images>
      <image>
<name>${docker.repository.name}:${project.version}</name>
        <alias>${project.artifactId}</alias>

        <build>
          <from>java:8-jre</from>
          <maintainer>sourabhh</maintainer>
          <assembly>
            <descriptor>docker-assembly.xml</descriptor>
          </assembly>
          <ports>
            <port>8080</port>
          </ports>
          <cmd>
            <shell>java -jar \
              /maven/${project.build.finalName}.jar server \
              /maven/docker-config.yml</shell>
          </cmd>
        </build>
        <run>
        <!-- To Do -->
        </run>
      </image>
```

```
    </images>
  </configuration>
</plugin>
```

以上的 Docker Maven 插件配置，创建了一个创建基于 JRE 8 (java:8-jre)的映像的 Dockerfile。这公开了端口 8080 和 8081。

接下来，我们将配置 docker-assembly.xml，它告诉插件应将哪些文件放入容器。它将被放置在 src/main/docker 目录下：

```
<assembly xmlns="http://maven.apache.org/plugins/maven-assembly-
plugin/assembly/1.1.2" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
  xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-
plugin/assembly/1.1.2 http://maven.apache.org/xsd/assembly-1.1.2.xsd">
  <id>${project.artifactId}</id>
  <files>
    <file>
      <source>{basedir}/target/${project.build.finalName}.jar</source>
      <outputDirectory>/</outputDirectory>
    </file>
    <file>
      <source>src/main/resources/docker-config.yml</source>
      <outputDirectory>/</outputDirectory>
    </file>
  </files>
</assembly>
```

上方的组装，把 service JAR 和 docker-config.yml 添加到生成的 Dockerfile 中。这个 Dockerfile 位于 target/docker/目录下。打开此文件，你将会发现类似于下面的内容：

```
FROM java:8-jre
MAINTAINER sourabhh
EXPOSE 8080
COPY maven /maven/
CMD java -jar \
```

```
  /maven/restaurant-service.jar server \
  /maven/docker-config.yml
```

前面的文件位于 `restaurant-service\target\docker\sousharm\restaurant-service\PACKT-SNAPSHOT\build`。Build 目录中还包含 maven 目录，其中包含在 `docker-assembly.xml` 中提到的所有东西。

让我们生成 Docker 映像：

```
mvn docker:build
```

此命令完成后，我们可以使用 Docker Images 或通过运行以下命令来验证本地存储库中的映像：

```
docker run -it -p 8080:8080 sourabhh/restaurant-service:PACKT-SNAPSHOT
```

使用`-it` 代替`-d` 在前台执行此命令。

使用 Maven 运行 Docker

若要使用 Maven 来执行 Docker 映像，我们需要在 `pom.xml` 中添加以下配置项。把`<run>`块放在我们在 `pom.xml` 文件 `docker-maven-plugin` 部分的 image 块下标记 *To Do* 的地方：

```
<properties>
  <docker.host.address>localhost</docker.host.address>
  <docker.port>8080</docker.port>
</properties>
...
<run>
  <namingStrategy>alias</namingStrategy>
  <ports>
    <port>${docker.port}:8080</port>
  </ports>
  <volumes>
    <bind>
      <volume>${user.home}/logs:/logs</volume>
    </bind>
```

```
    </volumes>
    <wait>
      <url>http://${docker.host.address}:${docker.port}/v1/
restaurants/1</url>
      <time>100000</time>
    </wait>
    <log>
      <prefix>${project.artifactId}</prefix>
      <color>cyan</color>
    </log>
  </run>
```

在这里，我们已经定义了运行餐馆服务的容器参数。我们映射了 Docker 容器的 8080 和 8081 端口到主机系统的端口，使我们能够访问此服务。同样，我们也已把容器的日志目录绑定到主机系统的<home>/logs 目录。

Docker Maven 插件通过轮询后台管理的 ping URL，直到它接收到一个回复，可以检测容器是否已完成启动。

请注意，如果你在 Windows 或者 Mac OS X 上使用 DockerToolbox 或 boot2docker，Docker 主机不是 localhost。可以通过执行 `docker-machine ip default` 检查 Docker 映像的 IP，它也在启动时显示。

Docker 容器已经准备就绪，用下面的命令来使用 Maven 启动它：

```
mvn docker:start .
```

使用 Docker 执行集成测试

启动和停止 Docker 容器可以通过将以下执行块绑定到在 `pom.xml` 中的 `docker-maven-plugin` 生命周期阶段来完成：

```
<execution>
  <id>start</id>
  <phase>pre-integration-test</phase>
  <goals>
    <goal>build</goal>
```

```
        <goal>start</goal>
      </goals>
    </execution>
    <execution>
      <id>stop</id>
      <phase>post-integration-test</phase>
      <goals>
        <goal>stop</goal>
      </goals>
    </execution>
```

我们现在将配置故障安全插件来用 Docker 执行集成测试，这使我们能够执行集成测试。我们在 `service.url` 标记中传递服务 URL，以便我们的集成测试可以使用它来执行测试。

我们会使用 `DockerIntegrationTest` 标志来标记我们的 Docker 集成测试。它的定义如下：

```
package com.packtpub.mmj.restaurant.resources.docker;

public interface DockerIntegrationTest {
    // Docker集成测试的标志
}
```

看看下面的集成插件代码。可以看到 `DockerIntegrationTest` 被配置为列入集成测试（故障保护插件），而它不包括在单元测试（Surefire 插件）中：

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-failsafe-plugin</artifactId>
  <version>2.18.1</version>
  <configuration>
    <phase>integration-test</phase>
    <includes>
      <include>**/*.java</include>
    </includes>
    <groups>com.packtpub.mmj.restaurant.resources.docker.
DockerIntegrationTest</groups>
    <systemPropertyVariables>
```

```
      <service.url>http://${docker.host.address}:${docker.port}/</
service.url>
    </systemPropertyVariables>
  </configuration>
  <executions>
    <execution>
      <goals>
        <goal>integration-test</goal>
      </goals>
    </execution>
  </executions>
</plugin>
  <plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.18.1</version>
  <configuration>
    <excludedGroups>com.packtpub.mmj.restaurant.resources.docker.
DockerIntegrationTest</excludedGroups>
  </configuration>
</plugin>
```

一个简单的集成测试看起来像这样：

```
@Category(DockerIntegrationTest.class)
public class RestaurantAppDockerIT {

    @Test
    public void testConnection() throws IOException {
        String baseUrl = System.getProperty("service.url");
        URL serviceUrl = new URL(baseUrl + "v1/restaurants/1");
        HttpURLConnection connection = (HttpURLConnection) serviceUrl.
openConnection();
        int responseCode = connection.getResponseCode();
        assertEquals(200, responseCode);
    }
}
```

可以用下面的命令来使用 Maven 执行集成测试：

```
mvn integration-test
```

把映像推送到注册表

在 docker-maven-plugin 下添加下列标记把 Docker 映像发布到 Docker Hub：

```
<execution>
  <id>push-to-docker-registry</id>
  <phase>deploy</phase>
  <goals>
    <goal>push</goal>
  </goals>
</execution>
```

可以利用以下配置为 maven-deploy-plugin 跳过 JAR 发布：

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-deploy-plugin</artifactId>
  <version>2.7</version>
  <configuration>
    <skip>true</skip>
  </configuration>
</plugin>
```

在 Docker Hub 发布一个 Docker 映像也需要一个用户名和密码：

```
mvn -Ddocker.username=<username> -Ddocker.password=<password> deploy
```

也可以将一个 Docker 映像推送到自己的 Docker 寄存处中。要做到这一点，请添加 docker.registry.name 标记，如下面的代码所示。例如，如果你的 Docker 寄存处在 xyz.domain.comon 的 4994 端口上是可用的，那么可以通过添加以下代码行来定义它：

```
<docker.registry.name>xyz.domain.com: 4994</docker.registry.name>
```

这完成了任务，我们不仅可以部署，而且也可以测试我们的 Dockerized 服务。

管理 Docker 容器

每个微服务都将有其自己的 Docker 容器。因此，我们将使用 *Docker Compose* 这个 Docker 容器管理器来管理我们的容器。

Docker Compose 将帮助我们指定容器的数量和如何执行这些容器。我们可以指定 Docker 映像、端口和每个容器到其他 Docker 容器的链接。

我们将在项目的目录创建一个文件称为 `docker-compose.yml` 的根，并向其中添加所有微服务的容器。我们会首先指定 Eureka 服务器，如下所示：

```
eureka:
  image: localhost:5000/sourabhh/eureka-server
  ports:
    - "8761:8761"
```

在这里，image 代表为 Eureka 服务器发布的 Docker 映像，ports 表示被用于执行 Docker 映像的主机和 Docker 主机之间的映射。

这将启动 Eureka 服务器并发布指定的端口用于外部访问。

现在，我们的服务可以使用这些容器（从属容器如 Eureka）。让我们看看 `restaurant-service` 是如何链接到相关的容器的。这很简单，只需使用 `links` 指令：

```
restaurant-service:
  image: localhost:5000/sourabhh/restaurant-service
  ports:
    - "8080:8080"
  links:
    - eureka
```

前面的 links 指令将更新 `restaurant-service` 容器中的`/etc/hosts` 文件，其中每行一个`restaurant-service`所依赖的服务（让我们假设`security`容器也被链接），例如：

```
192.168.0.22  security
192.168.0.31  eureka
```

如果你还没有设置 docker 本地寄存处，那么请首先做这个工作，以便问题较少或更顺畅地执行。

构建 docker 的本地寄存处：

```
docker run -d -p 5000:5000 --restart=always --name
registry registry:2
```

然后，对本地的映像执行 push 和 pull 命令：

```
docker push localhost:5000/sourabhh/restaurant-
service:PACKT-SNAPSHOT
docker-compose pull
```

最后，执行 docker-compose：

```
docker-compose up -d
```

一旦所有的微服务容器（服务和服务器）都被配置完成，我们就可以用单个命令启动所有 Docker 容器：

```
docker-compose up -d
```

这将启动在 Docker Compose 中配置的所有 Docker 容器。以下命令将列出它们：

```
docker-compose ps
Name                                        Command
              State          Ports
------------------------------------------------------------------
onlinetablereservation5_eureka_1          /bin/sh -c java -jar          ...
Up      0.0.0.0:8761->8761/tcp

onlinetablereservation5_restaurant-service_1  /bin/sh -c java -jar
...   Up      0.0.0.0:8080->8080/tcp
```

还可以使用以下命令检查 Docker 映像日志：

```
docker-compose logs
[36mrestaurant-service_1 | ←[0m2015-12-23 08:20:46.819  INFO 7 --- [pool-
3-thread-1] com.netflix.discovery.DiscoveryClient    : DiscoveryClient_
RESTAURANT-SERVICE/172.17
0.4:restaurant-service:93d93a7bd1768dcb3d86c858e520d3ce - Re-registering
apps/RESTAURANT-SERVICE
[36mrestaurant-service_1 | ←[0m2015-12-23 08:20:46.820  INFO 7 --- [pool-
```

```
3-thread-1] com.netflix.discovery.DiscoveryClient    : DiscoveryClient_
RESTAURANT-SERVICE/172.17
0.4:restaurant-service:93d93a7bd1768dcb3d86c858e520d3ce: registering
service...
[36mrestaurant-service_1 | ←[0m2015-12-23 08:20:46.917  INFO 7 --- [pool-
3-thread-1] com.netflix.discovery.DiscoveryClient    : DiscoveryClient_
RESTAURANT-SERVICE/172.17
```

参考资料

以下链接提供详细信息：

- **Netflix Ribbon**：https://github.com/Netflix/ribbon
- **Netflix Zuul**：https://github.com/Netflix/zuul
- **RabbitMQ**：https://www.rabbitmq.com/download.html
- **Hystrix**：https://github.com/Netflix/Hystrix
- **Turbine**：https://github.com/Netflix/Turbine
- **Docker**：https://www.docker.com/

小结

在这一章，我们已经了解了各种微服务管理功能——负载均衡、边缘服务器（网关）、电路断路器和监控。通过这一章的学习，现在应该知道如何实现负载均衡和路由，我们也学到了如何设置边缘服务器和配置。故障保护机制是在这一章已经学会的另一个重要部分。利用 Docker 或任何其他容器，可以简化部署。我们还使用 Maven 构建来演示了 Docker 和集成。

从测试的角度，我们对服务的 Docker 映像执行集成测试，还探讨了编写 RestTemplate 和 Netflix Feign 等客户端的方法。

在下一章中，我们将学会从身份验证和授权方面实现微服务的安全性，也将探讨微服务安全性的其他方面。

6
实现微服务的安全性

正如你所知，微服务是我们部署在处所内或云基础设施中的组件，微服务可以提供 API 或 web 应用程序。我们的示例应用程序 OTRS 提供的是 API。这一章将侧重如何使用 Spring Security 和 Spring OAuth2 来实现这些 API 的安全性，还会重点介绍 OAuth 2.0 基础知识，我们会使用 OAuth 2.0 来保护 OTRS 的 API。有关 REST API 安全保护的更多理解，可以参考 *Packt 出版的《RESTful Java Web Services Security》*一书，也可以参考 *Packt 出版的《Spring Security [Video]》*来学习更多的信息。我们也会了解跨起源请求网站(Cross Origin Request Site) 过滤器和跨站点脚本阻塞程序。

在这一章，我们将介绍以下主题：

- 启用安全套接字层（SSL）
- 身份验证和授权
- OAuth 2.0

启用安全套接字层

到目前为止，我们都在使用**超文本传输协议（Hyper Text Transfer Protocol，HTTP）**。HTTP 以纯文本形式传输数据，但以纯文本形式通过互联网传输数据，完全不是一个好主意。它方便了黑客的工作，使他们能够使用数据包嗅探器很容易地得到你的私人信息，如你的用户 ID、密码和信用卡详细信息。

我们肯定不想用户数据受到危害，所以我们将提供最安全的方式来访问我们的 web 应

用程序。因此，我们需要对最终用户和应用程序之间交换的信息进行加密。我们会使用**安全套接字层（Secure Socket Layer，SSL）**或**传输安全层（Transport Security Layer，TSL）**来加密数据。

SSL 是一种旨在为网络通信提供安全性（加密）的协议。HTTP 与 SSL 结合提供了 HTTP 的安全性实现，被称为**安全超文本传输协议（Hyper Text Transfer Protocol Secure）**，或**通过 SSL 的超文本传输协议（Hyper Text Transfer Protocol over SSL，HTTPS）**。HTTPS 可以确保被交换数据的隐私和完整性得到保护，它还确保被访问网站的真实性。此安全性围绕着托管应用程序的服务器、最终用户的机器，以及第三方信任存储服务器之间分布签名的数字证书。让我们看看这一过程是如何发生的：

1. 最终用户使用 web 浏览器将请求发送到 web 应用程序，例如 `http://twitter.com`。
2. 在接收到请求后，服务器使用HTTP代码302将浏览器重定向到`https://twitter.com`。
3. 最终用户的浏览器链接到 `https://twitter.com`，并且在响应中，服务器把其中包含数字签名的证书提供给最终用户的浏览器。
4. 最终用户的浏览器接收该证书，并将其发送到受信任的证书颁发机构（**Certificate Authority，CA**）进行验证。
5. 一旦证书获得根 CA 的验证，最终用户的浏览器和应用程序宿主服务器之间就建立了加密的通信。

安全HTTP通信

尽管 SSL 用加密和 web 应用程序身份验证的方式来保证安全性，但它不能保障不受网络钓鱼和其他攻击。专业黑客可以对使用 HTTPS 发送的信息进行解密。

现在，复习过 SSL 的基本知识，让我们为示例 OTRS 的项目实现它。我们不需要为所有的微服务都实现 SSL。除了新的微服务，即我们将在本章介绍的进行身份验证和授权的安全服务以外，所有其他微服务将都使用我们的代理或边缘服务器，Zuul 服务器被外部环境访问。

首先，我们会在边缘服务器上设置 SSL。我们需要有在嵌入式 Tomcat 中启用 SSL 所需的密钥存储库。为了演示，我们会使用自签名的证书，使用 Java keytool 用下面的命令生成密钥存储库，也可以使用任何其他工具来完成这件事：

```
keytool -genkey -keyalg RSA -alias selfsigned -keystore keystore.jks -ext
san=dns:localhost -storepass password -validity 365 -keysize 2048
```

它要求提供诸如名称、详细地址、组织等信息（见下面的截图）。

```
C:\dev\workspace\ms\online-table-reservation-6>keytool -genkey -keyalg RSA -alias selfsigned -keystor
What is your first and last name?
  [Unknown]:  localhost
What is the name of your organizational unit?
  [Unknown]:  org unit
What is the name of your organization?
  [Unknown]:  org
What is the name of your City or Locality?
  [Unknown]:  city
What is the name of your State or Province?
  [Unknown]:  state
What is the two-letter country code for this unit?
  [Unknown]:  CN
Is CN=localhost, OU=org unit, O=org, L=city, ST=state, C=CN correct?
  [no]:  yes

Enter key password for <selfsigned>
        (RETURN if same as keystore password):
Re-enter new password:

C:\dev\workspace\ms\online-table-reservation-6>
```

Keytool生成密钥

应注意以下几点以确保自签名证书的功能正常：

- 使用-ext 来定义**主题备用名称（Subject Alternative Names，SAN）**，还可以使用 IP（例如，san =ip:190.19.0.11）。早些时候，应用程序被部署到的机器的

主机名，被用作最常见的名称 **common name (CN)**。这个设置防止 `No name matching localhost found`（未找到匹配 localhost 的名称）的 `java.security.cert.CertificateException`。

- 可以使用浏览器或 OpenSSL 下载证书。利用 `keytool-importcert` 命令将新生成的证书添加到位于活动的 `JDK/JREhome` 目录 `jre/lib/security/cacerts` 中的 cacerts 密钥库。请注意，cacerts 密钥库的默认密码是 `changeit`。运行以下命令：

```
keytool -importcert -file path/to/.crt -alias <cert alias>
-keystore <JRE/JAVA_HOME>/jre/lib/security/cacerts -storepass
changeit
```

自签名的证书仅可以用于开发和测试目的。在生产环境中使用这些证书不能提供所需的安全。在生产环境中，要始终使用由受信任的颁发机构出具并签署的证书。安全地存储你的私钥。

现在，将生成的 `keystore.jks` 放入 OTRS 项目的 `src/main/resources` 目录，以及 `application.yml` 后，我们可以在边缘服务器的 `application.yml` 中更新此信息：

```
server:
   ssl:
      key-store: classpath:keystore.jks
      key-store-password: password
      key-password: password
   port: 8765
```

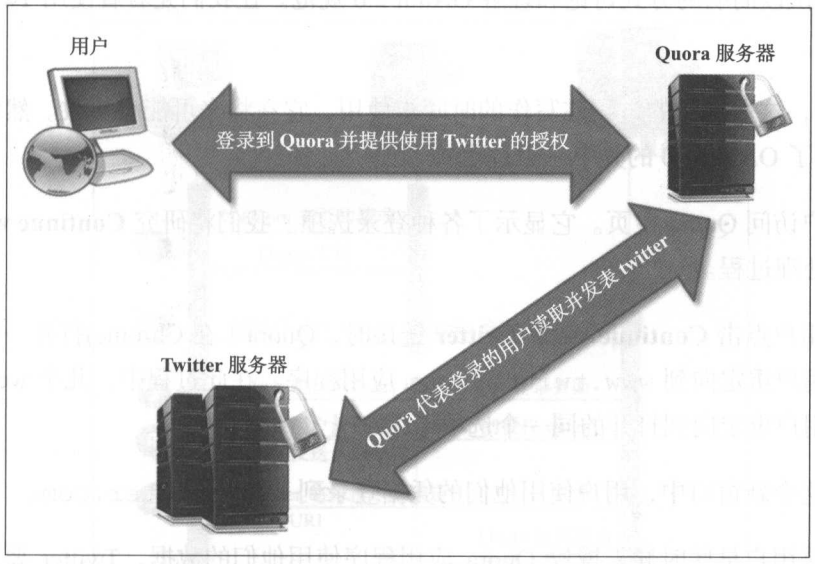
重建 Zuul 服务器 JAR 以使用 HTTPS。

在 Tomcat 7.0.66 以上和 8.0.28 以上版本中，密钥存储文件可以存储在前面的类路径中。对于较旧的版本，可以使用密钥存储文件的路径作为 `server:ssl:key-store` 值。

同样，也可以为其他微服务配置 SSL。

身份验证和授权

身份验证和授权事实上是对 web 应用程序提供的。在本节中，我们将讨论身份验证和授权。在过去几年中的新范例是 OAuth。我们将学习和使用 OAuth 2.0 来实现它。OAuth 是一种开放的授权机制，它在每个主要的 web 应用程序中都实现了。Web 应用程序可以通过实现 OAuth 标准访问彼此的数据，它已成为各种 web 应用程序来验证自己的最流行的方式。比如，在 `www.quora.com` 上，可以使用谷歌或 Twitter 的登录 ID 注册和登录。它也是用户友好的，因为客户端应用程序（例如 `www.quora.com`）不需要存储用户的密码。最终用户不需要多记一个用户 ID 和密码。



OAuth 2.0用法示例

OAuth 2.0

互联网工程任务组（**Internet Engineering Task Force，IETF**）规定了 OAuth 的标准和规格。在 OAuth 2.0 之前，OAuth 1.0a 曾经是最新的版本，它修复了 OAuth 1.0 中的会话固定安全缺陷。OAuth 1.0 和 1.0 a 与 OAuth 2.0 有很大的不同。OAuth 1.0 依赖于安全证书和通道绑定。OAuth 2.0 不支持安全认证和通道绑定，它完全工作在**传输安全层**（**Transport Security Layer, TSL**）上。因此，OAuth 2.0 不提供向后兼容性。

OAuth 的用法

- 如已讨论的，它可用于身份验证。你可能已经在各种应用中看到过它，如显示使用 Facebook 登录或使用 Twitter 登录的消息。
- 应用程序可以使用它从其他应用程序读取数据，如通过把一个 Facebook 构件集成到应用程序中，或在你的博客上有一个 Twitter 的回馈。
- 或者与前一点相反的过程：允许其他应用程序访问最终用户的数据。

OAuth 2.0 规范——简明详细信息

我们尝试以简明的方式讨论和理解 OAuth 2.0 规范。让我们先看看使用 Twitter 登录是如何工作的。

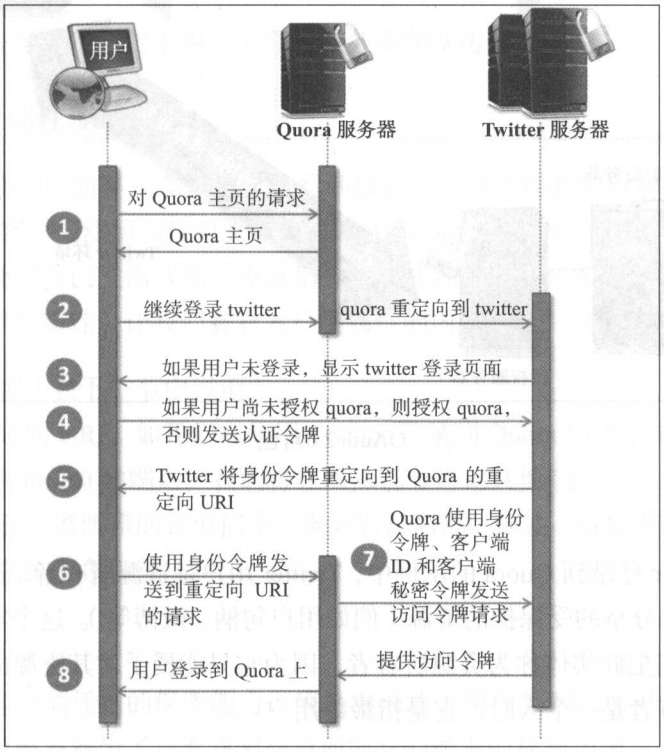
请注意，这里提到的过程在写作的时间被使用，它在将来可能会更改。然而，这一过程正确描述了 OAuth 2.0 的其中一个过程：

1. 用户访问 Quora 主页。它显示了各种登录选项。我们将研究 **Continue with Twitter** 的处理过程。
2. 当用户点击 **Continue with Twitter** 链接时，Quora（在 Chrome)打开一个新窗口，把用户重定向到 `www.twitter.com` 应用程序。在此过程中，几个 web 应用程序将用户重定向到打开的同一个选项卡/窗口上。
3. 在这个新窗口中，用户使用他们的凭据登录到 `www.twitter.com`。
4. 如果用户早些时候未授权 Quora 应用程序使用他们的数据，Twitter 要求用户授权 Quora 来访问用户信息的权限。如果用户已经授权 Quora，那么此步骤被跳过。
5. 经过正确的身份验证，Twitter 将用户和一个身份验证代码重定向到 Quora 的重定向 URI。
6. Quora 重定向 URI 在浏览器中输入时，Quora 发送客户端 ID、客户端秘密令牌和身份验证代码（Twitter 在步骤 5 中发送）到 Twitter。
7. Twitter 验证这些参数后，将访问令牌发送到 Quora。

8. 用户成功获取访问令牌后被登录到 Quora 上。

9. Quora 可能使用此访问令牌从 Quora 中获取用户信息。

你一定想知道 Twitter 是如何得到 Quora 的重定向 URI、客户端 ID 和秘密令牌的。Quora 充当一个客户端应用程序，而 Twitter 充当一台授权服务器。Quora 作为客户端，利用 Twitter 的 OAuth 实现，使用资源所有者（最终用户）的信息在 Twitter 上注册。Quora 在注册时提供一个重定向 URI，Twitter 把客户端 ID 和秘密令牌提供给 Quora。它以这种方式工作。在 OAuth 2.0 中，用户信息被称为用户资源。Twitter 提供一台资源服务器和授权服务器。我们将在下一节对这些 OAuth 术语进行更多的讨论。

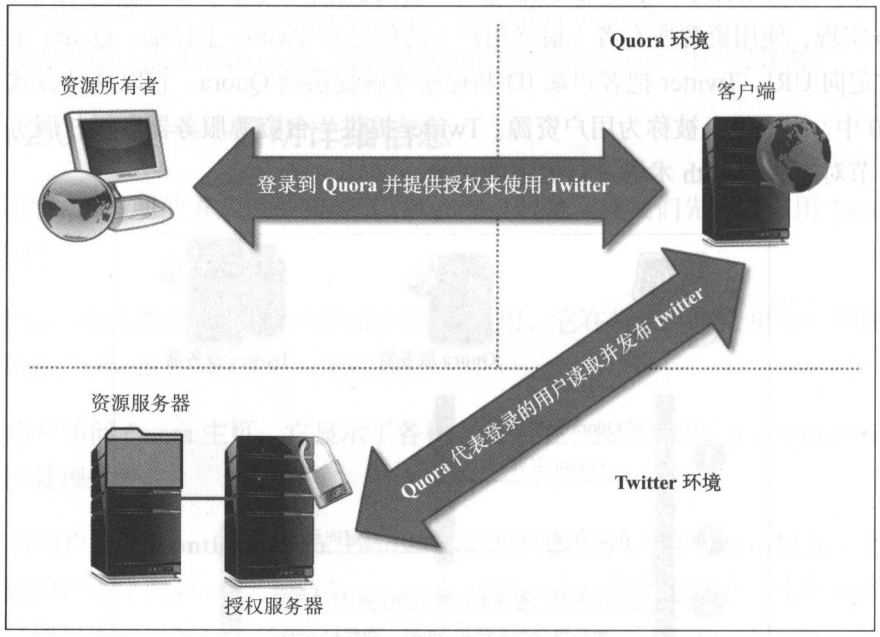


使用Twitter登录OAuth 2.0的示例过程

OAuth 2.0 角色

在 OAuth 2.0 规范中定义有 4 个角色：

- 资源所有者
- 资源服务器
- 客户端
- 授权服务器



OAuth 2.0角色

资源所有者

在使用 Twitter 登录到 Quora 的示例中，Twitter 用户是资源所有者。资源所有者是一个实体，拥有准备要分享的受保护的资源（例如用户句柄、微博等）。这个实体可以是应用程序或一个人。我们把此实体称为资源所有者，因为它只能授予对其资源的访问。规范还定义了，当资源所有者是一个人时，它是指最终用户。

资源服务器

资源服务器承载受保护的资源。它应该有能力使用访问令牌为访问这些资源的请求提供服务。对于使用 Twitter 登录到 Quora 的示例，Twitter 是资源服务器。

客户端

在使用 Twitter 登录到 Quora 的示例中，Quora 是客户端。客户端是代表资源所有者对资源服务器发出访问受保护资源请求的应用程序。

授权服务器

授权服务器为客户端应用程序提供不同的令牌，如访问令牌或刷新令牌，令牌只有在资源所有者对自己进行身份验证后才提供。

OAuth 2.0 不为资源服务器和授权服务器之间的交互提供任何规范。因此，授权服务器既可以与资源服务器在同一服务器上，也可以是一个单独的服务器。

单个授权服务器也可以用于为多个资源服务器颁发访问令牌。

OAuth 2.0 客户端注册

与授权服务器通信的客户端首先应注册到授权服务器才能获取资源的访问键。OAuth 2.0 规范没有指定客户端注册到授权服务器的方式。注册不需要客户端和授权服务器之间的直接通信，可以使用自己发出或第三方发出断言完成注册。授权服务器使用这些断言之一来获取所需的客户端属性。让我们看看客户端属性的内容：

- 客户端类型（在下一节中讨论）。
- 客户端重定向 URI，如我们在使用 Twitter 登录到 Quora 的示例中讨论的。这是一个用于 OAuth 2.0 的端点。我们将在**端点**部分讨论其他端点。
- 任何授权服务器所需的其他信息，例如客户名称、描述、徽标图像、联系方式及接受法律的条款和条件等。

客户端类型

基于它们对客户端凭据的保密能力，规范中描述了两种类型的客户端：保密和公开。客户端凭据是授权服务器为了与客户端交流而向它们颁发的秘密令牌。

保密客户端类型

这种客户端应用程序安全地保持密码和其他凭据或保密地维护它们。在使用 Twitter 登录到 Quora 的示例中，Quora 的应用程序服务器是安全的，并实现了限制的访问。因此，

它是保密性质的客户类型。只有 Quora 应用程序管理员拥有对客户端凭据访问的权限。

公开客户端类型

这种客户端应用程序不（*not*）安全地保持密码和其他凭据或保密地维护它们。移动设备或桌面电脑上的任何本机应用程序，或在浏览器上运行的应用程序是公共客户端类型的完美例子，因为这些应用程序在其中内嵌保持客户端凭据。黑客可以破解这些应用程序，客户端凭据可能会泄露。

客户端可以是一个分布式的基于组件的应用程序，例如，它可以同时有一个 web 浏览器组件和服务器端组件。在这种情况下，这两个组件会有不同的客户端类型和安全上下文。如果授权服务器不支持此类客户端，这些客户端应该把每个组件作为单独的客户端注册。

基于 OAuth 2.0 客户端类型，客户端可以有以下配置文件：

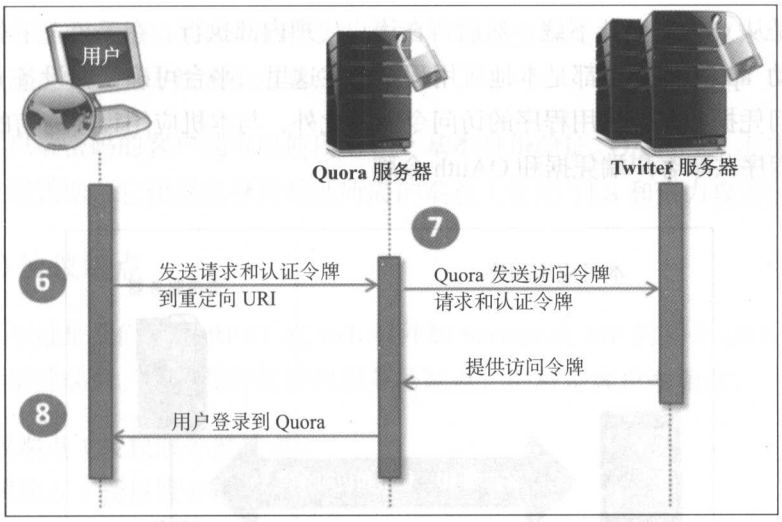
- Web 应用程序
- 基于用户代理的应用程序
- 本机应用程序

Web 应用程序

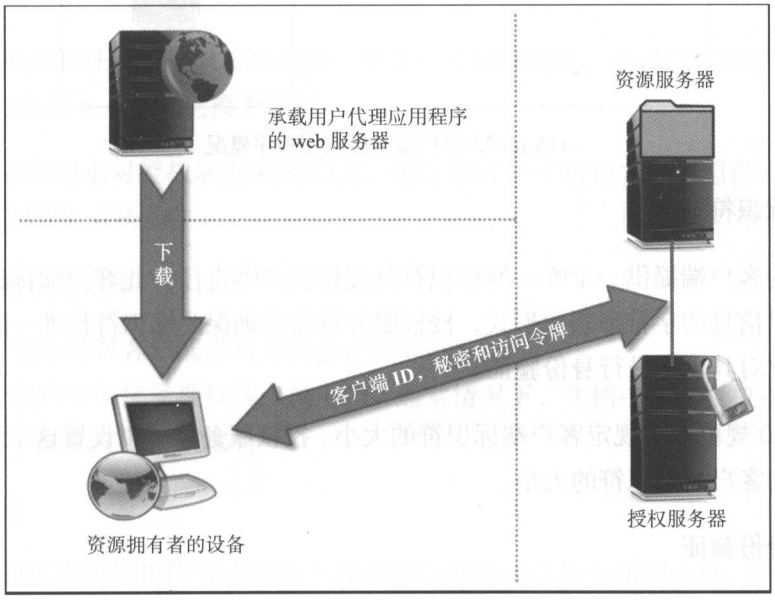
在使用 Twitter 登录到 Quora 的示例中，使用的 Quora web 应用程序是 OAuth 2.0 web 应用程序客户端配置文件的一个完美例子。Quora 是运行在 web 服务器上的保密客户端。资源所有者（最终用户）在他的设备（桌面电脑/平板电脑/手机）浏览器（用户代理）上使用 HTML 用户界面访问 Quora 上的应用程序（OAuth 2.0 客户端）。资源所有者不能访问客户端（Quora OAuth 2.0 客户端）的凭据和访问令牌，因为这些都存储在 web 服务器上。可以在 OAuth 2.0 示例流程图中看到这种行为。请参阅下页图所示的步骤 6 到 8。

基于用户代理的应用程序

基于用户代理的应用程序是公开客户端类型。在这里，虽然应用程序驻留在 web 服务器中，但资源所有者在用户代理（例如，web 浏览器）上下载它，然后再执行应用程序。在这里，驻留在资源所有者设备上的用户代理中下载的应用程序与授权服务器进行通信，资源所有者可以访问客户端凭据和访问令牌。游戏应用程序是这种应用程序配置文件的一个好例子。



OAuth 2.0客户端web应用程序配置文件

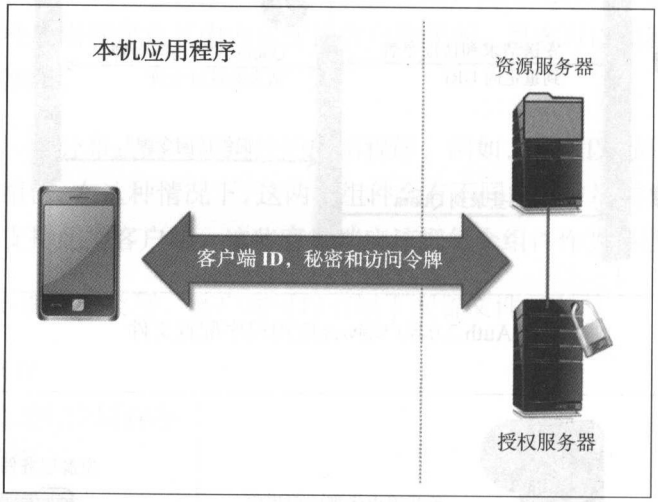


OAuth 2.0客户端用户代理应用程序概况

本机应用程序

本机应用程序类似基于用户代理的应用程序，但它们是在资源所有者的设备上安装和

执行，而不是从 web 服务器下载，然后再在用户代理内部执行。在手机上下载的许多本地客户端（移动 app）的类型都是本地应用程序。在这里，平台可确保在设备上的其他应用程序不能访问凭据和其他应用程序的访问令牌。此外，与本机应用程序通信的服务器不应与本机应用程序共享客户端凭据和 OAuth 令牌。



OAuth 2.0客户端本机应用程序概况

客户端标识符

向注册的客户端提供一个唯一的标识符是授权服务器责任。此客户端标识符是由注册的客户端提供信息的字符串表示形式，授权服务器需要确保此标识符是唯一的。授权服务器不应使用它对其自身进行身份验证。

OAuth 2.0 规范没有规定客户端标识符的大小。授权服务器可以设置这个大小，并应该记录它颁发的客户端标识符的大小。

客户端身份验证

授权服务器应基于其客户端类型验证客户端，应该确定适合并且符合安全要求的身份验证方法。在每个请求中，它应该只使用一种身份验证方法。

通常情况下，授权服务器使用一组客户端凭据，如客户端密码和一些密钥令牌来验证保密客户端的身份。

授权服务器可以与公共客户端建立一种客户端身份验证方法。然而，出于安全原因，它不得依赖此身份验证方法来确定客户端。

拥有客户端密码的客户端可以使用 HTTP 基本身份验证。OAuth 2.0 不建议在请求正文中发送客户端凭据，它建议在身份验证所需的端点上使用 TLS 和蛮力攻击保护。

OAuth 2.0 协议端点

端点只不过是我们使用 REST 或 web 组件如 Servlet 或 JSP 的一个 URI。OAuth 2.0 定义了三种类型的端点。其中两种是授权服务器端点，一种是客户端端点：

- 授权端点（授权服务器端点）
- 令牌端点（授权服务器端点）
- 重定向端点（客户端端点）

授权端点

此端点负责验证资源所有者的身份，并且一旦验证通过，获得授权许可（authorization grant）。我们将在下一节讨论授予许可。

授权服务器要求对授权端点采用 TLS，端点 URI 中不得包含片段组件，授权的端点必须支持 HTTP GET 方法。

规范没有规定以下内容：

- 授权服务器对客户端进行身份验证的方式。
- 客户端将如何接受授权端点的 URI。通常情况下，文档中包含授权端点的 URI，或者客户端在注册时获取它。

令牌端点

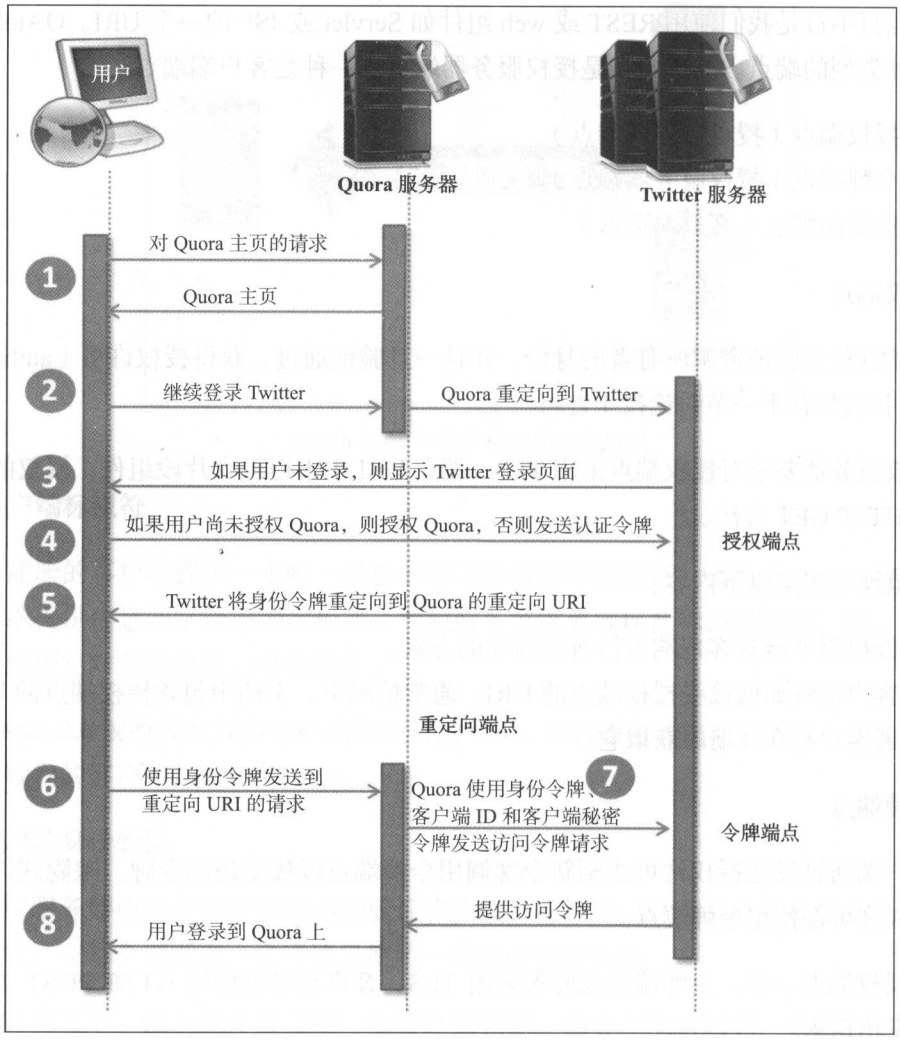
客户端通过发送授权许可或刷新令牌调用令牌端点以接受访问令牌。除隐式许可外的所有授权许可都使用令牌端点。

像授权端点一样，令牌端点也要求采用 TLS。客户端必须使用 HTTP POST 方法向令牌端点发出请求。

像授权端点一样，规范也没有指定客户端如何接收令牌端点的 URI。

重定向端点

一旦资源所有者和授权服务器之间的授权端点的交互完成，授权服务器就使用重定向端点把资源所有者的用户代理（例如 web 浏览器）重定向回客户端。客户端在注册时提供重定向端点，重定向端点必须是绝对 URI 并且不包含片段组件。



OAuth 2.0端点

OAuth 2.0 授权类型

客户端基于从资源所有者获得的授权，向授权服务器请求访问令牌。资源所有者以授权许可的形式给出授权。OAuth 2.0 定义了四种类型的授权许可：

- 授权代码许可
- 隐式许可
- 资源所有者密码凭据许可
- 客户端凭据许可

OAuth 2.0 还提供了一种扩展机制来定义额外的许可类型。可以在官方的 OAuth 2.0 规范中研究这个。

授权代码许可

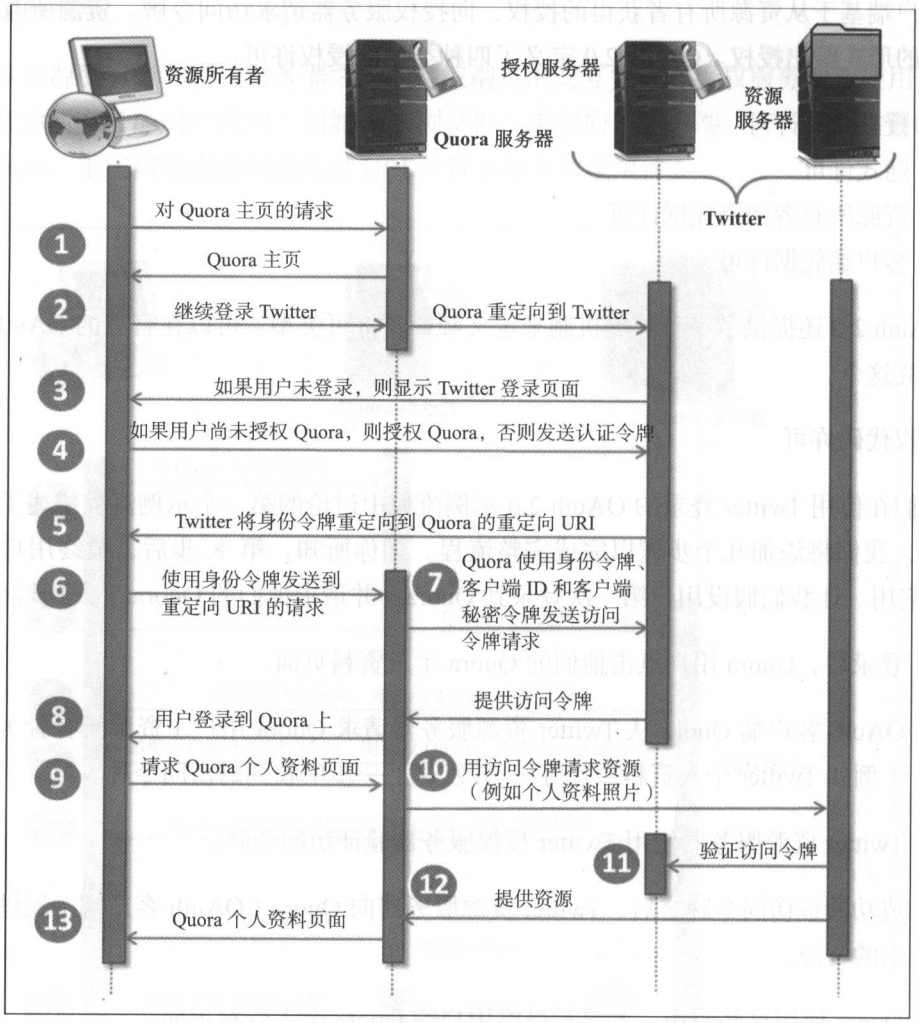
我们在使用 Twitter 登录的 OAuth 2.0 示例流程中讨论的第一个示例流程描述了授权代码许可，我们将添加几个步骤以完成完整流程。如你所知，第 8 步后，最终用户登录到 Quora 应用。让我们假设用户第一次登录到 Quora，并请求他们的 Quora 个人资料页面：

1. 登录后，Quora 用户点击他们的 Quora 个人资料页面。
2. OAuth 客户端 Quora 从 Twitter 资源服务器请求 Quora 用户（资源所有者）的资源（例如 Twitter 个人资料照片等），并发送上一步中收到的访问令牌。
3. Twitter 资源服务器使用 Twitter 授权服务器验证访问令牌。
4. 成功验证访问令牌之后，Twitter 资源服务器向 Quora（OAuth 客户端）提供了所要求的资源。
5. Quora 使用这些资源，并显示最终用户的 Quora 个人资料页面。

授权代码请求和响应

如果你查看授权代码流程的所有步骤（共 13 步），可以看到共有两个由客户端向授权服务器发出的请求，并且应答授权服务器提供了两个响应：一个请求-响应是针对验证令牌的，另一个请求-响应是针对访问令牌的。

让我们讨论一下用于每个请求和响应的参数。



OAuth 2.0授权代码许可流程

对授权端点 URI 的授权请求（步骤 4）：

参　数	必需/可选	描　述
response_type	必需	code（必须使用此值）。
client_id	必需	它表示注册时授权服务器发给客户端的 ID。
redirect_uri	可选	它表示注册时由客户端给出的重定向 URI。

续表

参　数	必需/可选	描　述
scope	可选	请求的范围。如果未提供，则授权服务器基于已定义的策略提供范围。
state	推荐	客户端使用此参数来保持客户端请求和（从授权服务器的）回调之间的状态。规范建议使用它来防止跨站点请求伪造攻击。

授权响应（步骤 5）：

参　数	必需/可选	描　述
code	必需	授权服务器生成的代码（授权码）。 代码在它生成后应该过期，建议的最大生存期是 10 分钟。 客户端使用此代码不得超过一次。 如果客户端使用它不止一次，那么请求必须被拒绝，并且以前基于此代码发出的所有令牌都应被撤销。 代码被绑定到客户端 ID 和重定向 URI。
state	必需	它表示注册时授权服务器发给客户端的 ID。

对令牌端点 URI 的令牌请求（步骤 7）：

参　数	必需/可选	描　述
grant_type	必需	authorization_code（必须使用此值）。
code	必需	从授权服务器接收到的代码（授权码）。
redirect_uri	必需	如果它被列入授权代码请求中，则此参数必需，并且值应该匹配。
client_id	必需	它表示注册时授权服务器发给客户端的 ID。

令牌响应（步骤 8）：

参　数	必需/可选	描　述
access_token	必需	由授权服务器颁发的访问令牌。
token_type	必需	授权服务器定义的令牌类型。在此基础上，客户端可以利用访问令牌。例如，bearer 或 mac。
refresh_token	可选	此令牌可以由客户端使用，使用授予的相同授权来获取新的访问令牌。
expires_in	推荐	表示以秒为单位的访问令牌生存期。值为 600 表示访问令牌的生存期是 10 分钟。如果在响应中不提供此参数，那么文档应突出显示访问令牌的生存期。

续表

参　数	必需/可选	描　述
scope	可选/必需	如果客户端请求的范围是相同的，则此参数是可选的。 如果访问令牌范围不同于客户端在请求中提供的那个，则此参数必需，通知客户端授予的访问令牌的实际范围。 如果客户端请求访问令牌时未提供范围，那么授权服务器应提供默认范围，或拒绝该请求，指示无效的范围。

错误响应：

参　数	必需/可选	描　述
error	必需	规范中定义的一个错误代码，例如，unauthorized_client、invalid_scope。
error_description	可选	错误的简短说明。
error_uri	可选	描述错误的错误页的 URI。

如果客户端在授权请求中传递了状态，则附加错误参数状态也在错误响应中发送。

隐式许可

我们在使用 Twitter 登录的 OAuth 2.0 示例流程中讨论的第一个示例流程描述了授权代码许可，我们将添加几个步骤以完成完整流程。如你所知，第 8 步后，最终用户登录到 Quora 应用。让我们假设用户第一次登录到 Quora，并请求他们的 Quora 个人资料页面：

1. 第 9 步：登录后，Quora 用户点击他们的 Quora 个人资料页面。
2. 第 10 步：OAuth 客户端 Quora 从 Twitter 资源服务器请求 Quora 用户（资源所有者）的资源（例如，Twitter 个人资料照片等），并发送上一步中收到的访问令牌。
3. 第 11 步：Twitter 资源服务器使用 Twitter 授权服务器验证访问令牌。
4. 第 12 步：成功验证访问令牌之后，Twitter 资源服务器向 Quora（OAuth 客户端）提供所要求的资源。
5. 第 13 步：Quora 使用这些资源，并显示最终用户的 Quora 个人资料页面。

隐式许可请求和响应

如果你查看授权代码流程的所有步骤（共 13 步），可以看到共有两个由客户端向授权

服务器发出的请求，并且应答授权服务器提供了两个响应：一个请求-响应是针对验证令牌的，另一个请求-响应是针对访问令牌的。

让我们讨论一下用于每个请求和响应的参数。

对授权端点 URI 的授权请求:

参　数	必需/可选	描　述
response_type	必需	Token（必须使用此值）。
client_id	必需	它表示注册时授权服务器发给客户端的 ID。
redirect_uri	可选	它表示注册时由客户端给出的重定向 URI。
scope	可选	请求的范围。如果未提供，则授权服务器基于已定义的策略提供范围。
state	推荐	客户端使用此参数来保持客户端请求和（从授权服务器的）回调之间的状态。规范建议使用它来防止跨站点请求伪造攻击。

访问令牌的响应：

参　数	必需/可选	描　述
access_token	必需	由授权服务器颁发的令牌。
token_type	必需	授权服务器定义的令牌类型。在此基础上，客户端可以利用访问令牌。例如，bearer 或 mac。
refresh_token	可选	此令牌可以由客户端使用，使用授予的相同授权来获取新的访问令牌。
expires_in	推荐	表示以秒为单位的访问令牌生存期。值为 600 表示访问令牌的生存期是 10 分钟。如果在响应中不提供此参数，那么文档应突出显示访问令牌的生存期。
scope	可选/必需	如果客户端请求的范围是相同的，则此参数是可选的。 如果访问令牌范围不同于客户端在请求中提供的那个，则此参数必需，通知客户端授予的访问令牌的实际范围。 如果客户端请求访问令牌时未提供范围，那么授权服务器应提供默认范围，或拒绝该请求，指示无效的范围。
state	可选/必需	如果状态是在客户端授权请求中传入的，则此参数必需。

错误响应：

参　数	必需/可选	描　述
error	必需	规范中定义的一个错误代码，例如，unauthorized_client、invalid_scope。

续表

参　数	必需/可选	描　述
error_description	可选	错误的简短说明。
error_uri	可选	描述错误的错误页的 URI。

如果客户端在授权请求中传递了状态，则附加错误参数状态也在错误响应中发送。

资源所有者密码凭据许可

我们在使用 Twitter 登录的 OAuth 2.0 示例流程中讨论的第一个示例流程描述了授权代码许可。我们将添加几个步骤以完成完整流程。如你所知，第 8 步后，最终用户登录到 Quora 应用。让我们假设用户第一次登录到 Quora，并请求他们的 Quora 个人资料页面：

1. 第 9 步：登录后，Quora 用户点击他们的 Quora 个人资料页面。
2. 第 10 步：OAuth 客户端 Quora 从 Twitter 资源服务器请求 Quora 用户（资源所有者）的资源（例如，Twitter 个人资料照片等），并发送上一步中收到的访问令牌。
3. 第 11 步：Twitter 资源服务器使用 Twitter 授权服务器验证访问令牌。
4. 第 12 步：成功验证访问令牌之后，Twitter 资源服务器向 Quora（OAuth 客户端）提供所要求的资源。
5. 第 13 步：Quora 使用这些资源，并显示最终用户的 Quora 个人资料页面。

资源所有者密码凭据许可请求和响应。

正如你在上一节中看到的，在授权代码流程的所有步骤（共 13 步）中，可以看到共有两个由客户端向授权服务器发出的请求，并且应答授权服务器提供了两个响应：一个请求-响应是针对验证令牌的，另一个请求-响应是针对访问令牌的。

让我们讨论一下用于每个请求和响应的参数。

对令牌端点 URI 的访问令牌请求:

参　数	必需/可选	描　述
grant_type	必需	password（必须使用此值）。
username	必需	资源所有者的用户名。

续表

参　数	必需/可选	描　述
password	必需	资源所有者密码。
scope	可选	请求的范围。如果未提供，则授权服务器基于已定义的策略提供范围。

访问令牌响应（步骤 8）：

参　数	必需/可选	描　述
access_token	必需	由授权服务器颁发的令牌。
token_type	必需	授权服务器定义的令牌类型。在此基础上，客户端可以利用访问令牌。例如，bearer 或 mac。
refresh_token	可选	此令牌可以由客户端使用，使用授予的相同授权来获取新的访问令牌。
expires_in	推荐	表示以秒为单位的访问令牌生存期。值为 600 表示访问令牌的生存期是 10 分钟。如果在响应中不提供此参数，那么文档应突出显示访问令牌的生存期。
可选参数	可选	额外的参数。

客户端凭据许可

我们在使用 Twitter 登录的 OAuth 2.0 示例流程中讨论的第一个示例流程描述了授权代码许可，我们将添加几个步骤以完成完整流程。如你所知，第 8 步后，最终用户登录到 Quora 应用。让我们假设用户第一次登录到 Quora，并请求他们的 Quora 个人资料页面：

1. 第 9 步：登录后，Quora 用户点击他们的 Quora 个人资料页面。
2. 第 10 步：OAuth 客户端 Quora 从 Twitter 资源服务器请求 Quora 用户（资源所有者）的资源（例如，Twitter 个人资料照片等），并发送上一步中收到的访问令牌。
3. 第 11 步：Twitter 资源服务器使用 Twitter 授权服务器验证访问令牌。
4. 第 12 步：成功验证访问令牌之后，Twitter 资源服务器向 Quora（OAuth 客户端）提供所要求的资源。
5. 第 13 步：Quora 使用这些资源，并显示最终用户的 Quora 个人资料页面。

客户端凭据许可请求和响应。

如果你查看授权代码流程的所有步骤（共 13 步），可以看到共有两个由客户端向授权服务器发出的请求，并且应答授权服务器提供了两个响应：一个请求-响应是针对验证令牌的，另一个请求-响应是针对访问令牌的。

让我们讨论一下用于每个请求和响应的参数。

对令牌端点 URI 的访问令牌请求：

参　数	必需/可选	描　述
grant_type	必需	client_credentials（必须使用此值）。
scope	可选	请求的范围。如果未提供，则授权服务器基于已定义的策略提供范围。

访问令牌响应:

参数	必需/可选	描述
access_token	必需	由授权服务器颁发的令牌。
token_type	必需	授权服务器定义的令牌类型。在此基础上，客户端可以利用访问令牌。例如，bearer 或 mac。
expires_in	推荐	表示以秒为单位的访问令牌生存期。值为 600 表示访问令牌的生存期是 10 分钟。如果在响应中不提供此参数，那么文档应突出显示访问令牌的生存期。

使用 Spring Security 的 OAuth 实现

OAuth 2.0 是保护 API 的一种方式。Spring Security 提供了 Spring Cloud Security 和 Spring Cloud OAuth2 组件来实现我们上面讨论的许可流程。

我们会再多创建一个服务，安全服务（security-service），它将控制身份验证和授权。

创建一个新的 Maven 项目，并按照下列步骤操作：

1. 在 pom.xml 中添加 Spring Security 和 Spring Security OAuth2 依赖项：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-security</artifactId>
</dependency>
```

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-oauth2</artifactId>
</dependency>
```

2. 在应用程序类中使用@EnableResourceServer 注解。这将允许此应用程序可以作为资源服务器工作。根据 OAuth 2.0 规格，@EnableAuthorizationServeris 是我们启用授权服务器要使用的另一个注解：

```
@SpringBootApplication
@RestController
@EnableResourceServer
public class SecurityApp {

    @RequestMapping("/user")
    public Principal user(Principal user) {
        return user;
    }

    public static void main(String[] args) {
        SpringApplication.run(SecurityApp.class, args);
    }

    @Configuration
    @EnableAuthorizationServer
    protected static class OAuth2Config extends
AuthorizationServerConfigurerAdapter {

        @Autowired
        private AuthenticationManager authenticationManager;

        @Override
        public void configure(AuthorizationServerEndpointsConfigurer
endpointsConfigurer) throws Exception {
          endpointsConfigurer.authenticationManager(authenticationManager);
        }
```

```
        @Override
        public void configure(ClientDetailsServiceConfigurer
clientDetailsServiceConfigurer) throws Exception {
  // 使用硬编码的内存中机制，因为它只是一个示例

            clientDetailsServiceConfigurer.inMemory()
            .withClient("acme")
            .secret("acmesecret")
            .authorizedGrantTypes("authorization_code", "refresh_
token", "implicit", "password", "client_credentials")
            .scopes("webshop");
        }
    }
}
```

3. 在 application.yml 中更新安全服务配置，如下面的代码所示：

   - server.contextPath：它表示上下文路径。
   - security.user.password：我们会为本演示使用硬编码的密码。在实际使用中可以重新配置它：

```
application.yml
info:
    component:
        Security Server

server:
    port: 9001
    ssl:
        key-store: classpath:keystore.jks
        key-store-password: password
        key-password: password
    contextPath: /auth

security:
    user:
        password: password
```

```
logging:
    level:
        org.springframework.security: DEBUG
```

现在我们有了安全服务器，我们会使用新的微服务 `api-service` 公开我们的 API，这将用于与外部应用程序和 UI 进行通信。

创建一个新的 Maven 项目，并按照下列步骤操作：

1. 在 `pom.xml` 中添加 Spring Security 和 Spring Security OAuth2 依赖项：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-undertow</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>

<dependency>
  <groupId>com.packtpub.mmj</groupId>
  <artifactId>online-table-reservation-common</artifactId>
  <version>PACKT-SNAPSHOT</version>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-security</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-oauth2</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
<dependency>
```

```
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-bus-amqp</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
  </dependency>
  <dependency>
    <groupId>org.apache.httpcomponents</groupId>
    <artifactId>httpclient</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <!-- Testing starter -->
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
  </dependency>
```

2. 在应用程序类中使用@EnableResourceServer 注解。这将允许此应用程序作为资源服务器：

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableCircuitBreaker
@EnableResourceServer
@ComponentScan({"com.packtpub.mmj.api.service", "com.packtpub.mmj.
common"})
public class ApiApp {

    private static final Logger LOG = LoggerFactory.
```

```
getLogger(ApiApp.class);

    static {
        //仅供本地主机测试
        LOG.warn("现在将禁用 SSL中的主机名检查，只在开发期间使用");
        HttpsURLConnection.setDefaultHostnameVerifier((hostname,
sslSession) -> true);
    }

    @Value("${app.rabbitmq.host:localhost}")
    String rabbitMqHost;

    @Bean
    public ConnectionFactory connectionFactory() {
        LOG.info("Create RabbitMqCF for host: {}", rabbitMqHost);
        CachingConnectionFactory connectionFactory = new CachingCo
nnectionFactory(rabbitMqHost);
        return connectionFactory;
    }

    public static void main(String[] args) {
        LOG.info("Register MDCHystrixConcurrencyStrategy");
        HystrixPlugins.getInstance().
registerConcurrencyStrategy(new MDCHystrixConcurrencyStrategy());
        SpringApplication.run(ApiApp.class, args);
    }
}
```

3. 在 application.yml 中更新 api-service 配置，如下面的代码所示：

- security.oauth2.resource.userInfoUri：它表示安全服务用户URI。

```
application.yml
info:
  component: API Service

spring:
  application:
    name: api-service
```

```
  aop:
    proxyTargetClass: true

server:
  port: 7771

security:
  oauth2:
    resource:
      userInfoUri: https://localhost:9001/auth/user

management:
  security:
    enabled: false
## 其他属性，比如Eureka、Logging等
```

现在我们有了安全服务器，我们会使用新的微服务 `api-service` 公开 API，这将用于与外部应用程序和 UI 进行通信。

现在让我们测试和探索它如何为不同的 OAuth 2.0 许可类型工作。

我们会使用 Chrome 浏览器的 `postman` 扩展来测试不同的流程。

授权码许可

我们将在浏览器中输入以下 URL，对授权码的一个请求如下所示：

```
https://localhost:9001/auth/oauth/authorize?response_
type=code&client_id=client&redirect_uri=http://localhost:7771/1&scope
=apiAccess&state=1234
```

在这里，我们提供客户端 ID（硬编码 client 是在默认情况下，我们在安全服务中注册的），重定向 URI、范围（在安全服务中的硬编码值 `apiAccess`）和状态。你一定想知道 `state` 参数的含义，它包含我们在响应中重新验证的随机数字，以防止跨站点请求伪造。

如果资源所有者（用户）尚未通过身份验证，它会要求输入用户名和密码。提供 `username`

作为用户名，password 作为密码，我们已经在安全服务中硬编码了这些值。

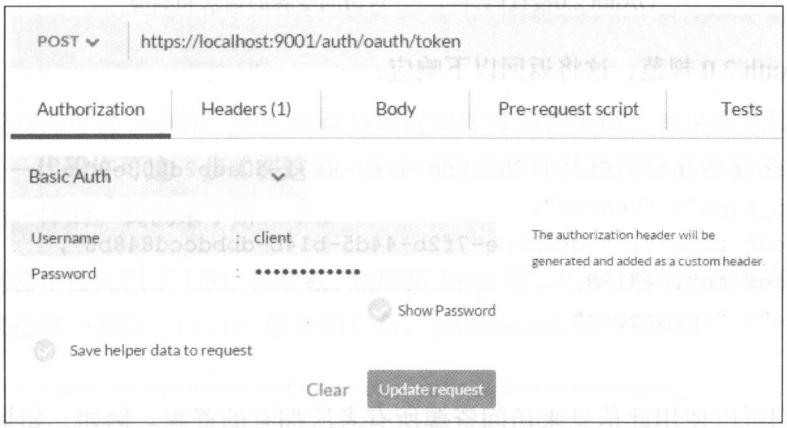
一旦登录成功，它将要求你（资源所有者）提供审批。

OAuth 2.0授权码许可——资源许可的审批

选中 **Approve**（**审批**）并在 **Authorize**（**授权**）上点击。此操作将把应用程序重定向到 http://localhost:7771/1?code=o8t4fi&state=1234。

正如你所看到的，它返回授权码和状态。

现在，我们将使用此代码来获取访问代码，将使用 Chrome 扩展 postman。首先，我们将使用用户名 client 和密码 clientsecret 添加授权标头，如下面的屏幕截图所示。



OAuth 2.0授权码许可 - 访问令牌请求 - 添加身份验证

这会将 Authorization 标头添加到请求中，值是 Basic Y2xpZW50OmNsaWVudHNlY3JldA==。

现在，我们会把其他几个参数添加到请求中，如下面的屏幕截图所示，然后提交该请求。

OAuth 2.0授权码许可——对访问令牌的请求和响应

根据 OAuth 2.0 规范，这将返回以下响应：

```
{
  "access_token": "6a233475-a5db-476d-8e31-d0aeb2d003e9",
  "token_type": "bearer",
  "refresh_token": "8d91b9be-7f2b-44d5-b14b-dbbdccd848b8",
  "expires_in": 43199,
  "scope": "apiAccess"
}
```

现在我们可以使用此信息来访问资源所有者所拥有的资源。例如，如果 `https://localhost:8765/api/restaurant/1` 表示 ID 为 1 的餐馆，那么它应该返回各餐馆的详细信息。

若没有访问令牌，如果我们输入这个 URL，它将返回错误 `Unauthorized`（未经授

权），以及 `Full authentication is required to access this resource`（访问该资源需要完整的身份验证）的消息。

现在，让我们使用访问令牌访问这个 URL，如下面的屏幕截图所示。

OAuth 2.0授权码许可——使用访问令牌进行API访问

正如你所看到的，我们在 **Authorization** 标头中加入了访问令牌。

现在，我们将研究隐式许可的实现。

隐式许可

除了代码许可这一步外，隐式许可与授权码许可非常类似。如果你从授权码许可中删除第一步——代码许可这一步（在这里，客户端应用程序从授权服务器接收授权令牌），剩下的步骤是一样的。我们来了解一下。

在浏览器中输入以下 URL 和参数，然后按 Enter 键。此外，请确保添加基本身份验证，如果被要求提供，则以 `client` 作为用户名，`password` 作为密码：

```
https://localhost:9001/auth/oauth/authorize?response_
type=token&redirect_uri=https://localhost:8765&scope=apiAccess&state=
553344&client_id=client
```

在这里，我们调用授权端点时使用以下的请求参数：响应类型、客户端 ID、重定向 URI、范围和状态。

当请求成功时，浏览器将被重定向到下面的 URL，并包含新的请求参数和值:

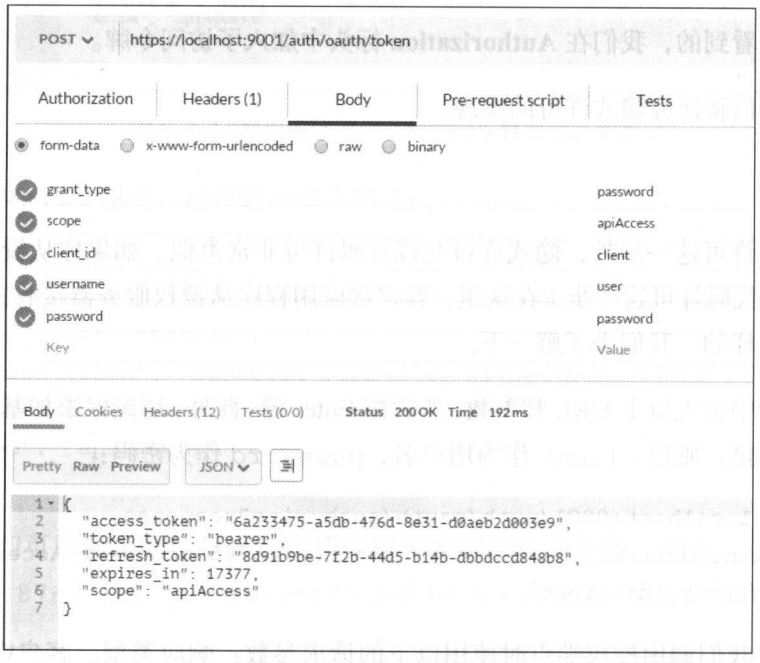
```
https://localhost:8765/#access_token=6a233475-a5db-476d-8e31-
d0aeb2d003e9&token_type=bearer&state=553344&expires_in=19592
```

在这里，我们收到令牌的 access_token、token_type、状态和到期期限。现在，我们可以使用这个访问令牌来访问 API，如同在授权码许可部分使用的那样。

资源所有者密码凭据许可

在这个许可中，我们在请求访问令牌时提供 username 和 password 作为参数，以及 grant_type、client 和 scope 参数。我们还需要使用客户端 ID 和密码来对请求进行身份验证。这些许可流程使用客户端应用程序代替浏览器，并且通常用于移动和桌面应用程序。

在下面的 postman 工具屏幕截图中，已使用基本身份验证和 client_id 及 password 添加了授权标头。



OAuth 2.0资源所有者密码凭据许可——访问令牌的请求和响应

一旦客户端收到访问令牌，它就可以用类似于它在授权码许可中的用法来使用。

客户端凭据许可

在这种流程中，客户端提供自己的凭据并获取访问令牌。它不使用资源所有者的凭据和权限。

可以在下面的屏幕截图中看到，我们直接输入令牌端点，只包含两个参数：`grant_type` 和 `scope`。使用 `client_id` 和 `client secret` 添加了授权标头。

OAuth 2.0客户端凭据许可——访问令牌的请求和响应

可以像授权码许可部分中解释的那样来使用访问令牌。

参考资料

更多的信息，请参阅以下链接：

- *RESTful Java Web Services Security, Packt Publishing*，Ren é Enr í quez, Andr é s Salazar C 著：`https://www.packtpub.com/application-development/restful-java-web-services-security`

- *Spring Security [Video], Packt Publishing*: `https://www.packtpub.com/application-development/spring-security-video`
- OAuth 2.0 授权框架: `https://tools.ietf.org/html/rfc6749`
- Spring Security: `http://projects.spring.io/spring-security`
- Spring OAuth2: `http://projects.spring.io/spring-security-oauth/`

小结

在本章中我们认识到，所有网络流量具备 TLS 层或 HTTPS 是多么的重要。我们在示例应用程序中加入一个自签名的证书。我想重申，对于生产应用程序，必须使用证书签发机构所提供的证书。我们还探讨了 OAuth 2.0 和各种 OAuth 2.0 授权流程的基础，不同的 OAuth 2.0 授权流程是使用 Spring Security 和 OAuth 2.0 实现的。在下一章中，我们将实现示例 OTRS 项目的用户界面，并将探讨所有组件是如何一起工作的。

7

利用微服务 Web 应用程序来使用服务

现在，开发了微服务之后，我们想看看如何可以通过 web 或移动应用程序使用**在线餐馆订位系统(OTRS)**所提供的服务。我们会使用 AngularJS/bootstrap 建立 web 应用程序(UI) 来构建 web 应用程序的原型。示例应用程序将显示此示例项目——一个小实用程序项目的数据和流程。此 web 应用程序也将是一个示例项目，并将独立运行。早期，web 应用程序被开发为单个 web 归档文件(.war 扩展名的文件)，它同时包含用户界面和服务器端代码。这样做的原因是开发过程相当简单，因为 UI 也使用 Java，包括 JSP、servlet、JSF 来开发。如今，UI 正在使用 JavaScript 独立开发。因此，这些 UI 应用程序也将部署为单独的微服务。在这一章，我们将探索如何开发这些独立的 UI 应用程序。我们将开发和实现没有登录和授权流程的 OTRS 示例应用程序。我们会部署一个功能非常有限的实现并涵盖高层次的 AngularJS 概念。要了解 AngularJS 的详细信息，你可以参考 Packt 出版的由 Chandermani 编写的《*AngularJS by Example*》一书。

在这一章，我们将介绍以下主题：

- AngularJS 框架概述
- 开发 OTRS 的功能
- 建立一个 web 应用程序(UI)

AngularJS 框架概述

现在既然我们准备好了 HTML5 web 应用程序的设置，可以浏览一下 AngularJS 的基本知识，这将帮助我们理解 AngularJS 代码。本节描述了对 AngularJS 高层次的理解，可以利

用它来了解示例应用程序，并利用 AngularJS 文档或通过参考其他 Packt 出版物来做进一步研究。

AngularJS 是一个客户端 JavaScript 框架。它有足够的灵活性来用作 **MVC**（**Model View Controller 模型视图控制器**）或 **MVVM**（**Model-View-ViewModel 模型-视图-视图模型**），它还使用依赖项注入模式提供内置的服务，如`$http`或`$log`。

MVC

MVC 是知名的设计模式。Struts 和 Spring MVC 是流行的例子，让我们看看它们是如何适合 JavaScript 环境的：

- **模型**：模型是 JavaScript 对象，其中包含应用程序数据。它们也表示应用程序的状态。
- **视图**：视图是表示层，其中包含 HTML 文件。在这里，可以显示来自模型的数据，并向用户提供交互式界面。
- **控制器**：可以在 JavaScript 中定义控制器，它包含应用程序逻辑。

MVVM

MVVM 是一种专门针对 UI 开发的架构设计模式。MVVM 被设计为使得双向数据绑定更容易。双向数据绑定提供了模型和视图之间的同步，当模型（数据）更改时，它会立即反映在视图上。同样，当用户更改视图上的数据时，它也反映在模型上。

- **模型**：这非常类似于 MVC，并包含业务逻辑和数据。
- **视图**：像 MVC 一样，它包含表示逻辑或用户界面。
- **视图模型**：视图模型包含在视图和模型之间绑定的数据。因此，它是视图和模型之间的一个接口。

模块

对于任何 AngularJS 应用程序，模块都是我们定义的第一个事物。模块是一个容器，它包含控制器、服务、过滤器等应用程序的不同部分。AngularJS 应用程序可以在单个模块或多个模块中编写，AngularJS 模块也可以包含其他的模块。

许多其他 JavaScript 框架都使用 main 方法来实例化和联系应用程序的不同部分。AngularJS 并没有 main 方法。由于下列原因，它使用模块作为入口点：

- **模块化**：可以按照功能或可重用的组件来划分和创建应用程序。
- **简单**：你可能遇到过复杂和规模庞大的应用程序代码，它们使维护和扩展非常困难。更多的好处是，AngularJS 使代码简单、可读性好并容易理解。
- **测试**：它使单元测试和端到端测试更加容易，因为你可以重写配置并只装载所需的模块。

每个 AngularJS 的应用程序都需要有单个模块来引导我们的应用程序。AngularJS 应用程序需要以下三个部分：

- **应用程序模块**：包含 AngularJS 模块的一个 JavaScript 文件（app.js），如下所示。

```
var otrsApp = AngularJS.module('otrsApp', [ ])
//[] 包含对其他模块的引用
```

- **加载 Angular 库和应用模块**：包含对 JavaScript 文件与其他 AngularJS 库的引用的 index.html 文件。

```
<script type="text/javascript" src="AngularJS/AngularJS.js"/>
<script type="text/javascript" src="scripts/app.js"/></script>
```

- **应用程序 DOM 配置**：这告诉 AngularJS 应该发生启动的 DOM 元素的位置。它可以用两种方式完成：
  - Index.html 文件，它还包含 HTML 元素（通常是<html>）与 app.js 中给定的值的 ng- app（AngularJS 指令）属性。AngularJS 指令包含 ng 前缀（AngularJS）：<html lang="en"ng-app="otrsApp"class="no-js">。
  - 如果你异步加载 JavaScript 文件，使用此命令：AngularJS.Bootstrap (document.documentElement, ['otrsApp']);。

AngularJS 模块除了其他组件，如控制器、服务、过滤器，等等，还有两个重要组成部分，config()和 run()。

- config()用于注册和配置模块，并且它只用于提供者和使用$injector 的常量。$injector 是一个 AngularJS 服务，我们将在下一节介绍提供程序和$injector。在这里不能使用实例，它在完全配置之前，会阻止使用服务。
- run()是用于在使用前面的配置方法创建$injector 后执行代码。这只供实例和常量使用。此处的提供程序不能用于避免在运行时配置。

提供程序和服务

让我们看看下面的代码：

```
.controller('otrsAppCtrl', function ($injector) {
var log = $injector.get('$log');
```

$log 是内置的 AngularJS 服务，它提供了日志记录 API。在这里，我们使用另一个内置的服务，$injector，它允许我们使用$log 服务。$injector 是控制器中的参数。AngularJS 用函数定义和正则表达式向调用方，也称控制器提供$injector 服务。这些都是 AngularJS 如何有效地使用依赖注入模式的例子。

AngularJS 使用了大量的依赖注入模式。AngularJS 使用注入器服务（$injector）来实例化和布线我们在 AngularJS 应用程序中使用的大部分对象。这个注入器创建两种类型的对象——服务和专门的对象。

为了简化起见，可以说我们（开发人员）定义服务。与此相反，专门的对象是控制器、过滤器、指令之类的 AngularJS 工件。

AngularJS 提供五种方法类型来告诉注入器如何创建服务对象——**提供程序**、**值**、工厂、**服务**和**常量**。

- 提供程序是核心和最复杂的方法类型，其他的方法都是在它上面添加的合成糖衣。我们通常避免使用提供程序，除非我们需要创建需要全局配置的可重用代码。
- 值和常量的方法类型的工作方式恰如其名，两者都不能有依赖关系。此外，两者的区别在于它们的用法，在配置阶段，不能使用值服务对象。
- 工厂和服务是使用最多的服务类型，它们是相似的类型。当我们想要生成 JavaScript 基元和函数时，我们使用工厂食谱。另一方面，当我们想要生成自定义的类型时，将使用服务。

现在我们对服务有了一些了解，可以说，服务有两种常见的用法——组织代码和在应用程序之间共享代码。服务是单例对象，它由 AngularJS 服务工厂懒惰地实例化。到目前为止，我们已经看到一些内建的 AngularJS 服务，如$injector、$log，等等。AngularJS 服务均以$符号作为前缀。

作用域

AngularJS 应用程序中有两种广泛使用的作用域类型：$rootScope 和$scope:

- $rootScope 是作用域层次结构中顶层的对象，它与全局的作用域关联。这意味着，附加到它上面的任何变量都能在任何地方使用，因此使用$rootScope 应当是经过深思熟虑的决定。
- 控制器把$scope 作为回调函数中的一个参数，它用于将数据从控制器绑定到视图，其作用域仅限于与其关联的控制器使用。

控制器

当控制器有一个$scope 作为参数，它由 JavaScript constructor 函数定义。控制器的主要目的是把数据结合到视图。控制器函数也用于编写业务逻辑——设置$scope 对象的初始状态并将行为添加到$scope。控制器的签名如下所示：

```
RestModule.controller('RestaurantsCtrl', function ($scope,
restaurantService) {
```

在这里，控制器是 RestModule 的一部分。控制器的名称是 RestaurantCtrl。$scope 和 restaurantService 都作为参数传递。

过滤器

过滤器的用途是格式化一个给定的表达式的值。下面的代码中我们定义了 datetime1 过滤器，它将日期作为参数并把值更改为 *dd MMM yyyy HH:mm* 的格式，如 *04 Apr 2016 04:13 PM*。

```
.filter('datetime1', function ($filter) {
    return function (argDateTime) {
        if (argDateTime) {
```

```
            return $filter('date')(new Date(argDateTime), 'dd MMM yyyy HH:mm a');
        }
        return "";
    };
});
```

指令

我们已经在模块一节看到，AngularJS 指令是带有 ng 前缀的 HTML 属性。下面是一些流行的指令：

- ng-app：此指令定义 AngularJS 应用程序
- ng-model：此指令将 HTML 表单输入绑定到数据
- ng-bind：此指令将数据绑定到 HTML 视图
- ng-submit：此指令提交 HTML 表单
- ng-repeat：此指令遍历集合

```
<div ng-app="">
    <p>Search: <input type="text" ng-model="searchValue"></p>
    <p ng-bind="searchedTerm"></p>
</div>
```

UI-Router

在**单页面应用程序(single page applications, SPAs)**中，页面只加载一次，用户通过不同的链接导航，而不刷新页面。这都是因为路由才有可能。路由是能使 SPA 导航感觉像普通网站的一种方法。因此，路由对于 SPA 是非常重要的。

AngularUI 团队构建了 UI-Router，它是一种 AngularJS 路由框架，不是核心 AngularJS 的一部分。当用户点击 SPA 中的任何链接时，UI-Router 不仅改变路由 URL，它还更改应用程序的状态。因为 UI-Router 还可以进行状态更改，可以在不更改 URL 的情况下更改页面的视图。由于应用程序状态由 UI-Router 管理，而使这变得可能。

如果我们把 SPA 当作一个状态机，那么其状态是应用程序的当前状态。当我们创建路由链接时，将在 HTML 链接标签使用 ui-sref 属性。链接中的属性 href 将从这生成并

指向在 app.js 中创建的应用程序的特定状态。

我们在HTML div 中使用 ui-view 属性来使用 UI-Router：例如，<div ui-view> 。

OTRS 功能的开发

正如你所知，我们正在开发 SPA。因此，一旦应用程序加载，你就可以执行所有操作而无须刷新页面。与服务器的所有交互都是使用 AJAX 调用执行的。现在，我们会使用第一节介绍的 AngularJS 的概念，会包括以下场景：

- 一个将显示餐馆列表的页面，这也将是我们的主页。
- 搜索餐馆。
- 餐馆详细信息和预订选项。
- 登录（不登录到服务器上，而是用于显示流程）。
- 预订确认。

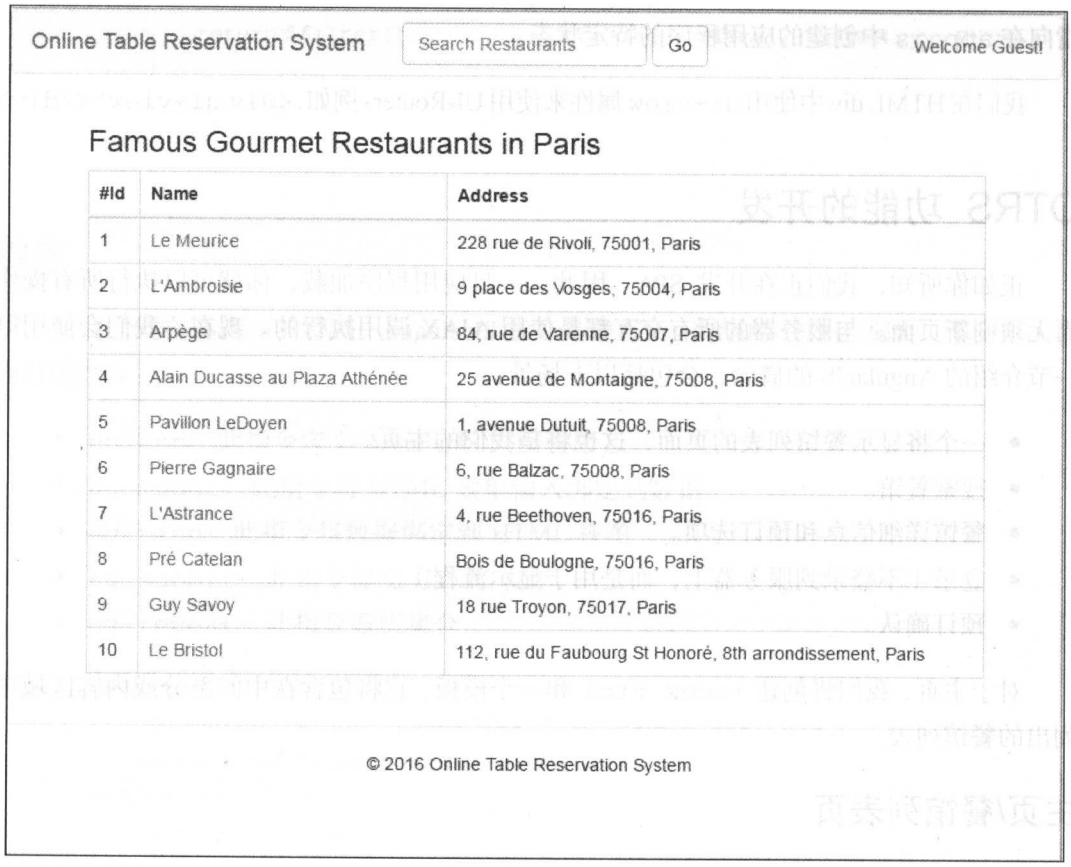
对于主页，我们将创建 index.html 和一个模板，它将包含在中间部分或内容区域中列出的餐馆列表。

主页/餐馆列表页

主页是任何 web 应用程序的主页面。为了设计主页，我们打算使用 Angular-UI 引导程序，而不是实际的引导程序。Angular-UI 是引导程序的 Angular 版本。主页将分为三个部分：

- 页眉部分将包含应用程序名称、搜索餐馆表单，以及在右上角的用户名称。
- 内容或中间部分将包含餐馆清单，它将用餐馆名称作为链接。此链接将指向餐馆详细信息和预订页面。
- 页脚部分将包含应用程序名称与版权标记。

你一定有兴趣在设计或实现主页之前查看它。因此，让我们首先来看看，一旦我们准备好了内容，它看起来的样子。



包含餐馆清单的OTRS主页

现在，来设计我们的主页，我们需要添加以下四个文件：

- `index.html`：我们的主 HTML 文件
- `app.js`：我们的主 AngularJS 模块
- `restaurants.js`：餐馆模块，它还包含餐馆 Angular 服务
- `restaurants.html`：将显示餐馆列表的 HTML 模板

index.html

首先，我们会把`./app/index.html`添加到我们的项目工作区。`Index.html`的内容将从这里开始描述。

我在代码中添加了注释，使代码更具可读性，并使它易于理解。

index.html 被分成许多部分，我们将讨论一些关键的部件。首先，我们将看到如何处理 Internet Explorer 浏览器的旧版本。如果想要针对版本大于 8 或 IE 9 起的 Internet Explorer 浏览器，那么我们需要添加以下块，这会阻止 JavaScript 的呈现，并把不含 js 的输出呈现给最终用户。

```
<!--[if lt IE 7]>      <html lang="en" ng-app="otrsApp" class="no-js
lt-ie9 lt-ie8 lt-ie7"> <![endif]-->
<!--[if IE 7]>         <html lang="en" ng-app="otrsApp" class="no-js
lt-ie9 lt-ie8"> <![endif]-->
<!--[if IE 8]>         <html lang="en" ng-app="otrsApp" class="no-js
lt-ie9"> <![endif]-->
<!--[if gt IE 8]><!--> <html lang="en" ng-app="otrsApp" class="no-js">
<!--<![endif]-->
```

然后，在添加几个 meta 标记和应用程序的标题后，我们还会定义重要的 meta 标记**视区（viewport）**。视区用于响应 UI 设计。

在内容属性中定义的 width 属性控制视区的大小。它可以设置为特定数量的像素宽度，如 width = 600 或特定的设备宽度值，即 100%比例的 CSS 像素屏幕的宽度。

initial-scale 属性控制第一次加载网页时的缩放级别。maximum-scale、minimum-scale 和 user-scalable 属性控制允许用户如何放大或缩小页面。

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

在接下来的几行中，我们将定义应用程序的样式表。我们将添加 HTML5 样板代码中的 normalize.css 和 main.css，也加入应用程序的自定义 CSS app.css。最后，我们添加 bootstrap 3 CSS。除了自定义的 app.css，其他 CSS 都被它引用。这些 CSS 文件都没有变化。

```
<link rel="stylesheet" href="bower_components/html5-boilerplate/dist/
css/normalize.css">
      <link rel="stylesheet" href="bower_components/html5-
boilerplate/dist/css/main.css">
```

```
        <link rel="stylesheet" href="public/css/app.css">
        <link data-require="bootstrap-css@*" data-server="3.0.0"
rel="stylesheet" href="//netdna.bootstrapcdn.com/bootstrap/3.0.0/css/
bootstrap.min.css" />
```

然后我们将使用 script 标记定义脚本。我们正在添加 modernizer、Angular、Angular-route 和我们自己开发的自定义 JavaScript 文件 app.js。我们已经讨论过 Angular 和 ngular-UI，app.js 将在下一节讨论。

modernizer 使 web 开发人员可以使用新的 CSS3 和 HTML5 功能，同时对并不支持它们的浏览器保持精细级别的控制权。基本上，modernizer 在页面加载到浏览器时执行下一代功能检测（检查这些功能的可用性），并报告结果。基于这些结果，可以检测浏览器中有哪些最新功能可用，可以基于此向最终用户提供一个接口。如果浏览器不支持一些功能，那么就把备用流程或 UI 提供给最终用户。

我们还使用 ui-bootstrap-tpls javascript 文件把用 JavaScript 写的引导程序模板添加进去。

```
        <script src="bower_components/html5-boilerplate/dist/js/
vendor/modernizr-2.8.3.min.js"></script>
        <script src="bower_components/angular/angular.min.js"></script>

        <script src="bower_components/angular-route/angular-route.min.js">
</script>
        <script src="app.js"></script>
      <script data-require="ui-bootstrap@0.5.0" data-semver="0.5.0"
src="http://angular-ui.github.io/bootstrap/ui-bootstrap-tpls-
0.6.0.js"></script>
```

我们还可以向 head 标记添加样式，如下所示。这个样式允许下拉菜单正常工作。

```
        <style>
            div.navbar-collapse.collapse {
                display: block;
                overflow: hidden;
                max-height: 0px;
                -webkit-transition: max-height .3s ease;
```

```
            -moz-transition: max-height .3s ease;
            -o-transition: max-height .3s ease;
            transition: max-height .3s ease;
        }
        div.navbar-collapse.collapse.in {
            max-height: 2000px;
        }
    </style>
```

在 body 标记中我们使用 ng-controller 属性定义应用程序的控制器，在页面加载时，它把应用程序到 Angular 的名称告诉控制器。

```
<body ng-controller="otrsAppCtrl">
```

然后，我们定义主页的 header 部分。在 header 部分，我们将定义应用程序的标题，Online Table Reservation System。此外，我们还将定义用来搜索餐馆的搜索表单。

```
<!-- BEGIN HEADER -->
    <nav class="navbar navbar-default" role="navigation">

        <div class="navbar-header">
            <a class="navbar-brand" href="#">
                Online Table Reservation System
            </a>
        </div>
        <div class="collapse navbar-collapse" ng-
class="!navCollapsed && 'in'" ng-click="navCollapsed = true">
            <form class="navbar-form navbar-left" role="search"
ng-submit="search()">
                <div class="form-group">
                    <input type="text" id="searchedValue" ng-
model="searchedValue" class="form-control" placeholder="Search
Restaurants">
                </div>
                <button type="submit" class="btn btn-default" ng-
click="">Go</button>
            </form>
    <!-- END HEADER -->
```

然后，在下一部分，即中间部分，包括我们实际上绑定不同的视图，它使用实际的内容注释做标记。在 div 中的 ui-view 属性从 Angular 动态地获取其内容，如餐馆详细信息、餐馆列表，等等。我们还在中间部分添加了一个警告对话框和微调框，它们将在有必要时显示。

```
        <div class="clearfix"></div>
        <!-- 开始容器 -->
        <div class="page-container container">
            <!-- 开始内容 -->
            <div class="page-content-wrapper">
                <div class="page-content">
                    <!-- 开始实际内容 -->
                    <div ui-view class="fade-in-up"></div>
                    <!-- 结束实际内容 -->
                </div>
            </div>
            <!-- 结束内容 -->
        </div>
        <!-- 加载微调框 -->
        <div id="loadingSpinnerId" ng-show="isSpinnerShown()"
style="top:0; left:45%; position:absolute; z-index:999">
            <script type="text/ng-template" id="alert.html">
                <div class="alert alert-warning" role="alert">
                <div ng-transclude></div>
                </div>
            </script>
            <uib-alert type="warning" template-url="alert.
html"><b>Loading...</b></uib-alert>
        </div>
        <!-- 结束容器 -->
```

index.html 的最后一部分是页脚。可以添加你想要的任何内容，在这里，我们只添加静态内容和版权文本。

```
        <!-- 开始页脚 -->
        <div class="page-footer">
            <hr/><div style="padding: 0 39%">&copy; 2016 Online Table
```

```
Reservation System</div>
        </div>
        <!-- 结束页脚 -->
    </body>
</html>
```

app.js

app.js 是我们的主应用程序文件。因为我们在 index.html 中定义了它，所以只要 index.html 被调用，它就会被加载。

我们需要小心地不要把路由（URI）与 REST 端点混合。路由表示 SPA 的状态/视图。

由于我们使用了边缘服务器（代理服务器），一切都将通过它来访问，包括我们的 REST 端点。外部应用程序，包括用户界面将使用边缘服务器主机访问应用程序。可以在一些全局常量文件中配置它，然后在任何需要的地方使用它。这将允许你在一个单独的地方配置 REST 主机并在其他地方使用它。

```
'use strict';
/*
此调用初始化我们的应用程序并注册所有模块，作为数组中第二个参数传递。
*/
var otrsApp = angular.module('otrsApp', [
    'ui.router',
    'templates',
    'ui.bootstrap',
    'ngStorage',
    'otrsApp.httperror',
    'otrsApp.login',
    'otrsApp.restaurants'
])
/*
  然后我们定义了默认路由 /restaurants
*/
        .config([
```

```
            '$stateProvider', '$urlRouterProvider',
            function ($stateProvider, $urlRouterProvider) {
                $urlRouterProvider.otherwise('/restaurants');
            }])
/*
  此函数控制应用程序的流程并处理事件。
*/
        .controller('otrsAppCtrl', function ($scope, $injector,
restaurantService) {
            var controller = this;

            var AjaxHandler = $injector.get('AjaxHandler');
            var $rootScope = $injector.get('$rootScope');
            var log = $injector.get('$log');
            var sessionStorage = $injector.get('$sessionStorage');
            $scope.showSpinner = false;
/*
  当用户搜索任何一家餐馆时，此函数就被调用。它使用Angular餐馆服务，我们将在下一节中搜
索给定的搜索字符串时定义它。
*/
            $scope.search = function () {
                $scope.restaurantService = restaurantService;
                restaurantService.async().then(function () {
                    $scope.restaurants = restaurantService.
search($scope.searchedValue);
                });
            }
/*
  当状态更改时，新的控制器基于视图和配置控制流程，而现有的控制器被销毁。发生destroy事
件时，此函数被调用。
*/
            $scope.$on('$destroy', function destroyed() {
                log.debug('otrsAppCtrl destroyed');
                controller = null;
                $scope = null;
            });
```

```
            $rootScope.fromState;
            $rootScope.fromStateParams;
            $rootScope.$on('$stateChangeSuccess', function (event,
toState, toParams, fromState, fromStateParams) {
                $rootScope.fromState = fromState;
                $rootScope.fromStateParams = fromStateParams;
            });

            // 实用程序方法
            $scope.isLoggedIn = function () {
                if (sessionStorage.session) {
                    return true;
                } else {
                    return false;
                }
            };

            /* 微调框状态 */
            $scope.isSpinnerShown = function () {
                return AjaxHandler.getSpinnerStatus();
            };

        })
/*
  此对象加载时，此函数被执行。在这里我们正在设置为根作用域定义的用户对象。
*/
        .run(['$rootScope', '$injector', '$state', function
($rootScope, $injector, $state) {
            $rootScope.restaurants = null;
            // 自引用
            var controller = this;
            // 注入外部的引用
            var log = $injector.get('$log');
            var $sessionStorage = $injector.get('$sessionStorage');
            var AjaxHandler = $injector.get('AjaxHandler');

            if (sessionStorage.currentUser) {
```

```
            $rootScope.currentUser = $sessionStorage.currentUser;
        } else {
            $rootScope.currentUser = "Guest";
            $sessionStorage.currentUser = ""
        }
    }])
```

restaurants.js

restaurants.js 表示我们的应用程序 Angular 服务，我们将把它用于餐馆。我们知道服务有两种常见的用途——组织代码和在应用程序之间共享代码。因此，我们已经创建了一个餐馆服务，它将用在不同的模块，如搜索、列表、详细信息，等等。

服务是单例对象，由 AngularJS 服务工厂懒惰地实例化。

下面这段代码初始化餐馆服务模块，并加载必需的依赖项。

```
angular.module('otrsApp.restaurants', [
    'ui.router',
    'ui.bootstrap',
    'ngStorage',
    'ngResource'
])
```

在配置信息中，我们使用 UI-Router 定义了 otrsApp.restaurants 模块的路由和状态。

首先我们通过传递 JSON 对象来定义 restaurants 状态，这个 JSON 对象包含指向路由 URI 的 URL、指向显示 restaurants 状态的 HTML 模板的模板 URL，以及将处理 restaurants 视图上的事件的控制器。

在 restaurants 视图（route-/restaurants）之上，也定义了一个嵌套的状态 restaurants.profileis，它将表示具体的餐馆。例如，/restaurant/1 会打开并显示 Id 为 1 的餐馆的概要资料（详细信息）页。用户在 restaurants 模板中的某个链接上单击时，此状态将会被调用。在此 ui sref ="restaurants.profile ({id: rest.id})"中，rest 代表从 restaurantsview 获取的 restaurant 对象。

请注意，状态名称是'restaurants.profile'，这告诉 AngularJS UI Router，概要资料是一种 restaurants 状态的嵌套状态。

```
.config([
    '$stateProvider', '$urlRouterProvider',
    function ($stateProvider, $urlRouterProvider) {
        $stateProvider.state('restaurants', {
            url: '/restaurants',
            templateUrl: 'restaurants/restaurants.html',
            controller: 'RestaurantsCtrl'
        })
            // 餐馆显示页面
            .state('restaurants.profile', {
                url: '/:id',
                views: {
                    '@': {
                        templateUrl: 'restaurants/restaurant.html',
                        controller: 'RestaurantCtrl'
                    }
                }
            });
}])
```

在接下来的这段代码中，我们使用 Angular 工厂服务类型来定义餐馆服务。餐馆服务加载时使用 REST 调用从服务器获取餐馆的列表。它提供了一个列表、搜索餐馆操作和餐馆数据。

```
.factory('restaurantService', function ($injector, $q) {
    var log = $injector.get('$log');
    var ajaxHandler = $injector.get('AjaxHandler');
    var deffered = $q.defer();
    var restaurantService = {};
    restaurantService.restaurants = [];
    restaurantService.orignalRestaurants = [];
    restaurantService.async = function () {
        ajaxHandler.startSpinner();
        if (restaurantService.restaurants.length === 0) {
```

```
                ajaxHandler.get('/api/restaurant')
                        .success(function (data, status, headers, config) {
                            log.debug('Getting restaurants');
                            sessionStorage.apiActive = true;
                            log.debug("if Restaurants --> " +
restaurantService.restaurants.length);
                            restaurantService.restaurants = data;
                            ajaxHandler.stopSpinner();
                            deffered.resolve();
                        })
                        .error(function (error, status, headers, config) {
                            restaurantService.restaurants = mockdata;
                            ajaxHandler.stopSpinner();
                            deffered.resolve();
                        });
                return deffered.promise;
            } else {
                deffered.resolve();
                ajaxHandler.stopSpinner();
                return deffered.promise;
            }
        };
        restaurantService.list = function () {
            return restaurantService.restaurants;
        };
        restaurantService.add = function () {
            console.log("called add");
            restaurantService.restaurants.push(
                    {
                        id: 103,
                        name: 'Chi Cha\'s Noodles',
                        address: '13 W. St., Eastern Park, New County, Paris',
                    });
        };
        restaurantService.search = function (searchedValue) {
            ajaxHandler.startSpinner();
            if (!searchedValue) {
```

```
                if (restaurantService.orignalRestaurants.length > 0) {
                    restaurantService.restaurants =
restaurantService.orignalRestaurants;
                }
                deffered.resolve();
                ajaxHandler.stopSpinner();
                return deffered.promise;
            } else {
                ajaxHandler.get('/api/restaurant?name=' + searchedValue)
                        .success(function (data, status, headers, config) {
                            log.debug('Getting restaurants');
                            sessionStorage.apiActive = true;
                            log.debug("if Restaurants --> " +
restaurantService.restaurants.length);
                            if (restaurantService.
orignalRestaurants.length < 1) {
                                restaurantService.
orignalRestaurants = restaurantService.restaurants;
                            }
                            restaurantService.restaurants = data;
                            ajaxHandler.stopSpinner();
                            deffered.resolve();
                        })
                        .error(function (error, status, headers, config) {
                            if (restaurantService.
orignalRestaurants.length < 1) {
                                restaurantService.
orignalRestaurants = restaurantService.restaurants;
                            }
                            restaurantService.restaurants = [];
                            restaurantService.restaurants.push(
                                    {
                                        id: 104,
                                        name: 'Gibsons - Chicago Rush St.',
                                        address: '1028 N. Rush
St., Rush & Division, Cook County, Paris'
                                    });
```

```
                restaurantService.restaurants.push(
                    {
                        id: 105,
                        name: 'Harry Caray\'s Italian Steakhouse',
                        address: '33 W. Kinzie
St., River North, Cook County, Paris',
                    });
                ajaxHandler.stopSpinner();
                deffered.resolve();
            });
        return deffered.promise;
    }
};
return restaurantService;
})
```

在 restaurants.js 模块的下一部分，我们将为路由配置中定义的餐馆和 restaurants.profile 状态添加两个控制器。这两个控制器是 RestaurantsCtrl 和 RestaurantCtrlthat，它们分别处理 restaurants 状态和 restaurants.profiles 状态。

RestaurantsCtrl 很简单，因为它使用餐馆服务的列表方法来加载餐馆的数据。

```
.controller('RestaurantsCtrl', function ($scope, restaurantService) {
    $scope.restaurantService = restaurantService;
    restaurantService.async().then(function () {
        $scope.restaurants = restaurantService.list();
    });
})
```

RestaurantCtrl 负责显示一个给定 ID 的餐馆的详细信息，它也负责在显示的餐馆上执行预订操作。当我们设计餐馆详细信息页面与预订选项时，将使用此控件。

```
.controller('RestaurantCtrl', function ($scope, $state,
$stateParams, $injector, restaurantService) {
    var $sessionStorage = $injector.get('$sessionStorage');
    $scope.format = 'dd MMMM yyyy';
    $scope.today = $scope.dt = new Date();
```

```
        $scope.dateOptions = {
            formatYear: 'yy',
            maxDate: new Date().setDate($scope.today.getDate() + 180),
            minDate: $scope.today.getDate(),
            startingDay: 1
        };

        $scope.popup1 = {
            opened: false
        };
        $scope.altInputFormats = ['M!/d!/yyyy'];
        $scope.open1 = function () {
            $scope.popup1.opened = true;
        };
        $scope.hstep = 1;
        $scope.mstep = 30;

        if ($sessionStorage.reservationData) {
            $scope.restaurant = $sessionStorage.reservationData.
restaurant;
            $scope.dt = new Date($sessionStorage.reservationData.tm);
            $scope.tm = $scope.dt;
        } else {
            $scope.dt.setDate($scope.today.getDate() + 1);
            $scope.tm = $scope.dt;
            $scope.tm.setHours(19);
            $scope.tm.setMinutes(30);
            restaurantService.async().then(function () {
                angular.forEach(restaurantService.list(), function
(value, key) {
                    if (value.id === parseInt($stateParams.id)) {
                        $scope.restaurant = value;
                    }
                });
            });
        }
        $scope.book = function () {
```

```
                var tempHour = $scope.tm.getHours();
                var tempMinute = $scope.tm.getMinutes();
                $scope.tm = $scope.dt;
                $scope.tm.setHours(tempHour);
                $scope.tm.setMinutes(tempMinute);
                if ($sessionStorage.currentUser) {
                    console.log("$scope.tm --> " + $scope.tm);
                    alert("Booking Confirmed!!!");
                    $sessionStorage.reservationData = null;
                    $state.go("restaurants");
                } else {
                    $sessionStorage.reservationData = {};
                    $sessionStorage.reservationData.restaurant = $scope.
restaurant;
                    $sessionStorage.reservationData.tm = $scope.tm;
                    $state.go("login");
                }
            }
        })
```

我们也在 `restaurants.js` 模块中添加了一些过滤器来设置日期和时间格式。这些过滤器对输入数据执行以下格式化操作：

- `date1`：以'MMM dd yyyy'格式返回输入的日期，例如 13-Apr-2016
- `time1`：以'HH:mm:ss'格式返回输入的时间，例如 11:55:04
- `dateTime1`：以'dd MMM yyyy HH:mm:ss'格式返回输入的日期和时间，例如 13-Apr-2016 11:55:04

在下面的代码片段中，我们已经应用了这三个过滤器：

```
    .filter('date1', function ($filter) {
        return function (argDate) {
            if (argDate) {
                var d = $filter('date')(new Date(argDate), 'dd MMM yyyy');
                return d.toString();
            }
            return "";
```

```
        };
    })
    .filter('time1', function ($filter) {
        return function (argTime) {
            if (argTime) {
                return $filter('date')(new Date(argTime), 'HH:mm:ss');
            }
            return "";
        };
    })
    .filter('datetime1', function ($filter) {
        return function (argDateTime) {
            if (argDateTime) {
                return $filter('date')(new Date(argDateTime), 'dd MMM yyyy
HH:mm a');
            }
            return "";
        };
    });
```

restaurants.html

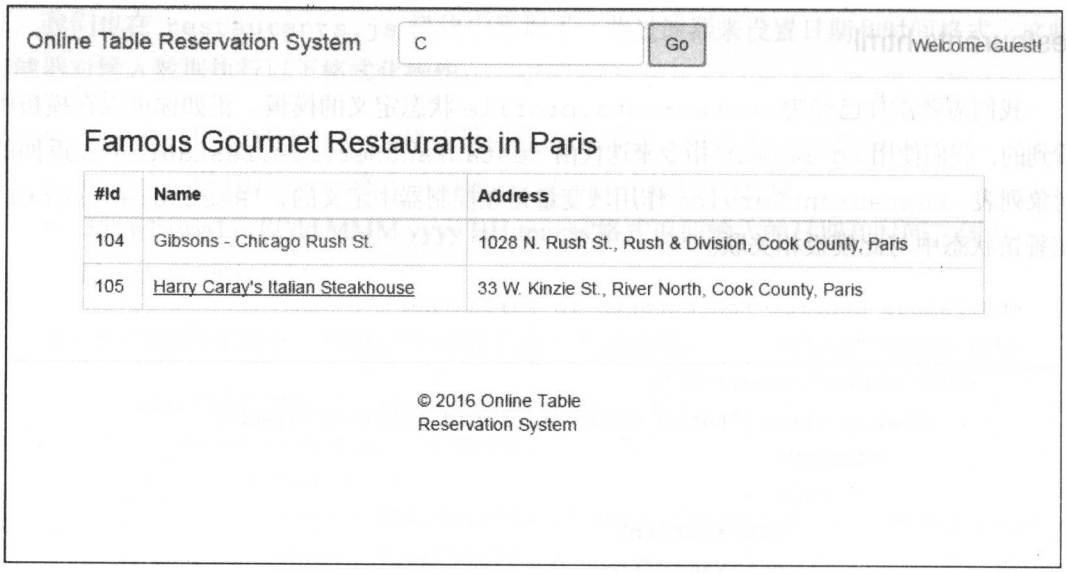
我们需要添加已经为 restaurants.profile 状态定义的模板。正如你可以在模板中看到的，我们使用 ng-repeat 指令来迭代由 restaurantService.restaurants 返回的对象列表。RestaurantService 作用域变量是在控制器中定义的，'RestaurantsCtrl' 在餐馆状态中与此模板相关联。

```
<h3>Famous Gourmet Restaurants in Paris</h3>
<div class="row">
    <div class="col-md-12">
        <table class="table table-bordered table-striped">
            <thead>
                <tr>
                    <th>#Id</th>
                    <th>Name</th>
                    <th>Address</th>
                </tr>
```

```
            </thead>
            <tbody>
                <tr ng-repeat="rest in restaurantService.restaurants">
                    <td>{{rest.id}}</td>
                    <td><a ui-sref="restaurants.profile({id: rest.id})">
{{rest.name}}</a></td>
                    <td>{{rest.address}}</td>
                </tr>
            </tbody>
        </table>
    </div>
</div>
```

搜索餐馆

我们已在主页 index.html 的 header 部分中添加搜索表单，使我们能够搜索餐馆。搜索餐馆功能将使用如前所述的相同文件。它利用 app.js（搜索表单处理程序）、restaurants.js（餐馆服务）和 restaurants.html 来显示搜索到的记录。

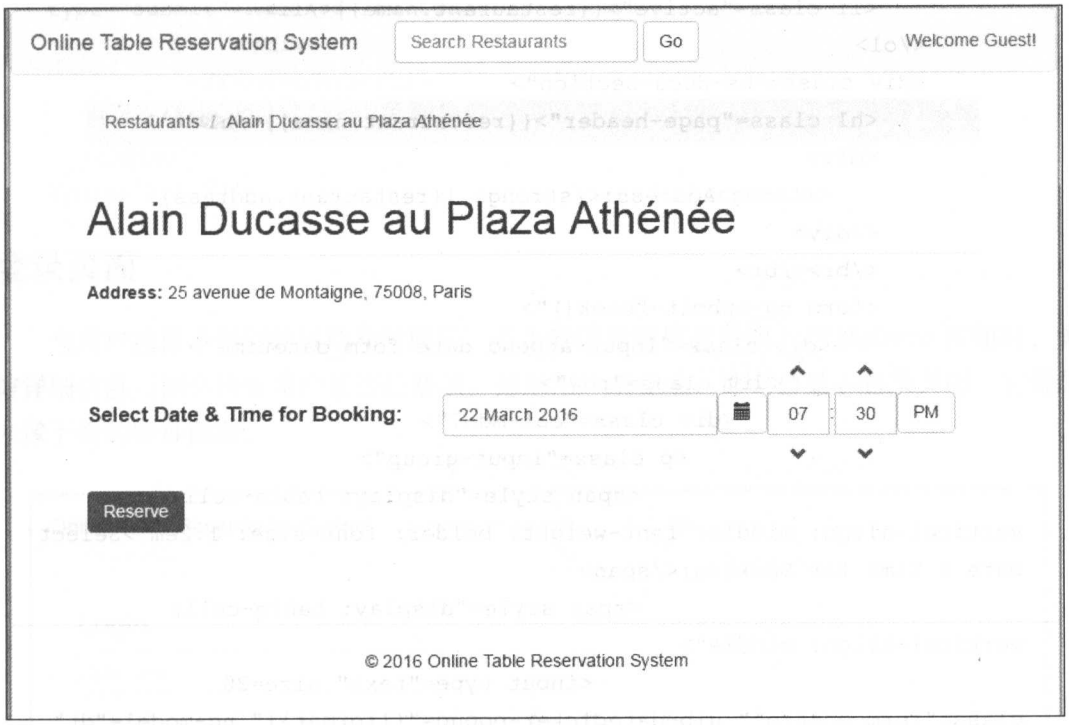


OTRS主页与餐馆清单

餐馆详细信息与预订选项

餐馆详细信息与预订选项将是内容区域（页面的中间部分）的一部分。这将包含顶部的痕迹导航，以及链接到餐馆清单页面的 Restaurants，后面跟着餐馆的名称和地址。最后一部分将包含预订部分，其中包含日期时间选择框和预订按钮。

此页面将看起来像下面的屏幕截图。



餐馆详细信息页面与预订选项

在这里，我们将使用与 `restaurants.js` 中声明的同一个餐馆服务。如状态 `restaurants.profile` 所述，唯一的变化就是模板。此模板将使用 `restaurant.html` 来定义。

restaurant.html

正如你所看到的，痕迹导航使用由 `ui-sref` 属性定义的餐馆路由。在此模板中设计

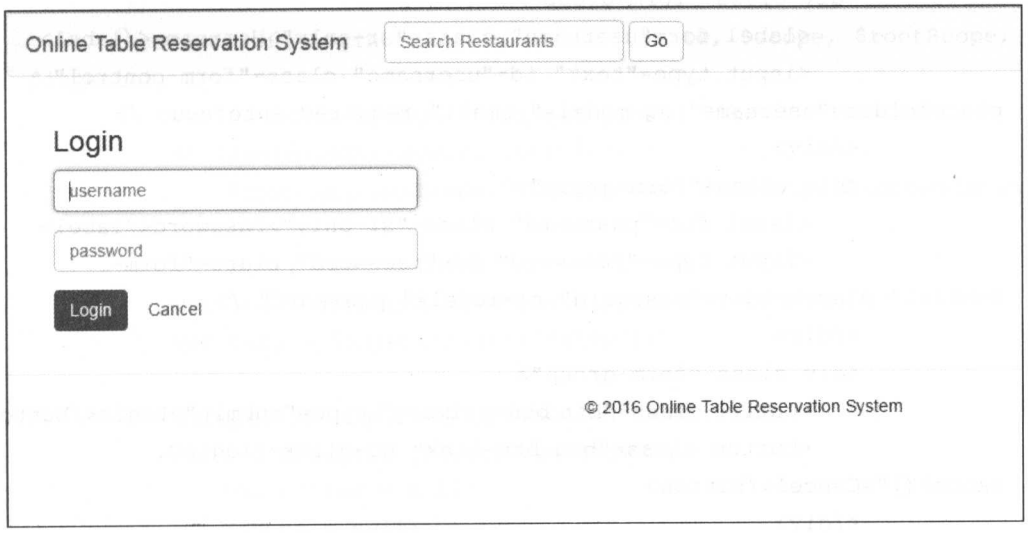
的预订表格调用 book()函数，此函数在控制器 RestaurantCtrl 中用表单提交的指令 ng-submiton 定义。

```
<div class="row">
<div class="row">
    <div class="col-md-12">
        <ol class="breadcrumb">
            <li><a ui-sref="restaurants">Restaurants</a></li>
            <li class="active">{{restaurant.name}}</li>
        </ol>
        <div class="bs-docs-section">
            <h1 class="page-header">{{restaurant.name}}</h1>
            <div>
                <strong>Address:</strong> {{restaurant.address}}
            </div>
            </br></br>
            <form ng-submit="book()">
                <div class="input-append date form_datetime">
                    <div class="row">
                        <div class="col-md-7">
                            <p class="input-group">
                                <span style="display: table-cell;
vertical-align: middle; font-weight: bolder; font-size: 1.2em">Select
Date & Time for Booking:</span>
                                <span style="display: table-cell;
vertical-align: middle">
                                    <input type="text" size=20
class="form-control" uib-datepicker-popup="{{format}}" ng-model="dt"
is-open="popup1.opened" datepicker-options="dateOptions" ng-
required="true" close-text="Close" alt-input-formats="altInputFormats"
/>
                                </span>
                                <span class="input-group-btn">
                                    <button type="button" class="btn
btn-default" ng-click="open1()"><i class="glyphicon glyphicon-
calendar"></i></button>
                                </span>
```

```
                        <uib-timepicker ng-model="tm" ng-
change="changed()" hour-step="hstep" minute-step="mstep"></uib-
timepicker>
                        </p>
                    </div>
                </div></div>
            <div class="form-group">
                <button class="btn btn-primary"
type="submit">Reserve</button>
            </div>
        </form></br></br>
    </div>
  </div>
</div>
```

登录页面

当用户选择了预订的日期和时间后，点击**餐馆详细信息**页面上的 **Reserve** 按钮时，**餐馆详细信息**页面会检查用户是否已登录。如果用户未登录，那么会显示登录页面。它看起来像下面的屏幕截图。



登录页面

我们不从服务器上对用户进行身份验证。相反，我们只把用户名称填充在会话存储和实现工作流中的根作用域中。

一旦用户登录成功，用户将被重定向回相同的预订页面并保持原来的状态，然后用户可以继续处理预订。**Login** 页面基本上使用如下两个文件：`login.html` 和 `login.js`。

login.html

`Login.html` 模板由两个输入字段（用户名和密码），以及 **Login** 按钮和 **Cancel** 链接组成。**Cancel** 链接重置表单，而 **Login** 按钮提交登录表单。

在这里，我们使用 `LoginCtrl` 和 `ng-controller` 指令。**Login** 表单使用调用 `LoginCtrl` 的 `submit` 函数的 `ng-submit` 指令提交。输入的值首先使用 `ng-model` 指令收集，然后使用其各自的属性-`_email` 和 `_password` 提交。

```
<div ng-controller="LoginCtrl as loginC" style="max-width: 300px">
    <h3>Login</h3>
    <div class="form-container">
        <form ng-submit="loginC.submit(_email, _password)">
            <div class="form-group">
                <label for="username" class="sr-only">Username</label>
                <input type="text" id="username" class="form-control"
placeholder="username" ng-model="_email" required autofocus />
            </div>
            <div class="form-group">
                <label for="password" class="sr-only">Password</label>
                <input type="password" id="password" class="form-
control" placeholder="password" ng-model="_password" />
            </div>
            <div class="form-group">
                <button class="btn btn-primary" type="submit">Login</button>
                <button class="btn btn-link" ng-click="loginC.
cancel()">Cancel</button>
            </div>
        </form>
```

```
    </div>
</div>
```

login.js

登录模块是在 login.js 中定义的，它包含并加载使用此模块功能的依赖项。状态 login 是在 config 函数的帮助下定义的，此函数会取得包含 url、控制器和 templateUrl 属性的 JSON 对象。

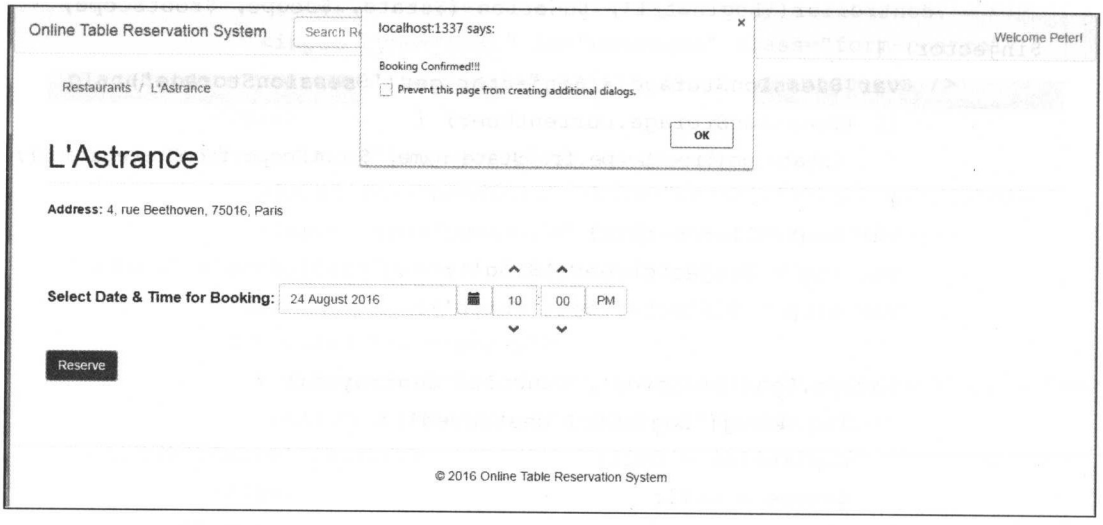
在控制器里面，我们定义取消和提交操作，它们是从 login.html 模板调用的。

```
angular.module('otrsApp.login', [
    'ui.router',
    'ngStorage'
])
        .config(function config($stateProvider) {
            $stateProvider.state('login', {
                url: '/login',
                controller: 'LoginCtrl',
                templateUrl: 'login/login.html'
            });
        })
        .controller('LoginCtrl', function ($state, $scope, $rootScope,
$injector) {
            var $sessionStorage = $injector.get('$sessionStorage');
            if ($sessionStorage.currentUser) {
                $state.go($rootScope.fromState.name, $rootScope.fromStateParams);
            }
            var controller = this;
            var log = $injector.get('$log');
            var http = $injector.get('$http');

            $scope.$on('$destroy', function destroyed() {
                log.debug('LoginCtrl destroyed');
                controller = null;
                $scope = null;
            });
```

```
            this.cancel = function () {
                $scope.$dismiss;
                $state.go('restaurants');
            }
            console.log("Current --> " + $state.current);
            this.submit = function (username, password) {
                $rootScope.currentUser = username;
                $sessionStorage.currentUser = username;
                if ($rootScope.fromState.name) {
                    $state.go($rootScope.fromState.name,
$rootScope.fromStateParams);
                } else {
                    $state.go("restaurants");
                }
            };
        });
```

预订确认

一旦用户登录成功并点击 **Reservation** 按钮，餐馆控制器就显示确认警告框，如以下屏幕截图所示。



餐馆详细信息页面与预订确认

设置 web 应用程序

我们打算为用户界面应用程序开发使用最新的技术栈，我们将使用 Node.js 和 **npm（Node.js 包管理器）**，它们为开发服务器端 JavaScript web 应用程序提供开放源码的运行时环境。

建议把本节通读一遍，它将向你介绍 JavaScript 构建工具和环境。然而，如果你了解 JavaScript 构建工具或不想研究它们，则可以跳过。

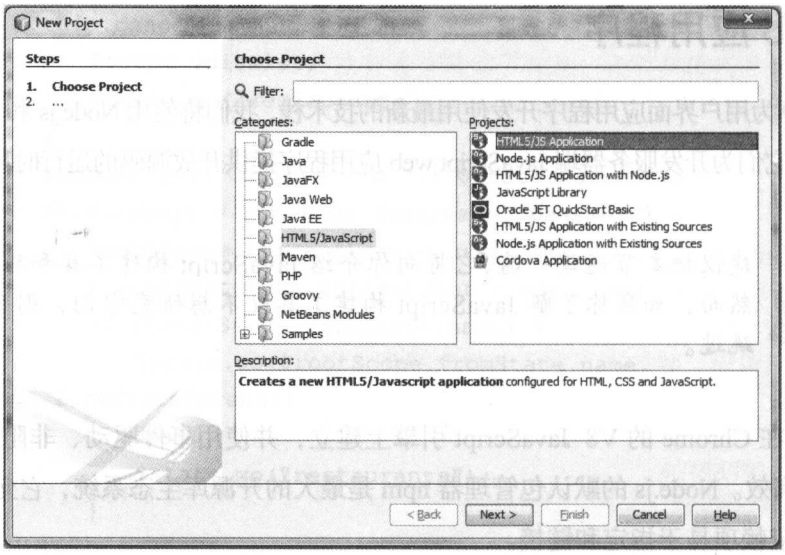
Node.js 在 Chrome 的 V8 JavaScript 引擎上建立，并使用事件驱动、非阻塞 I/O，这使得它轻便而高效。Node.js 的默认包管理器 npm 是最大的开源库生态系统，它允许安装 Node 程序并使得依赖项易于指定和链接。

1. 如果还没有安装 npm，我们需要首先安装它。它是一个先决条件。可以检查位于 `https://docs.npmjs.com/getting-started/installing-node` 的链接来安装 npm。

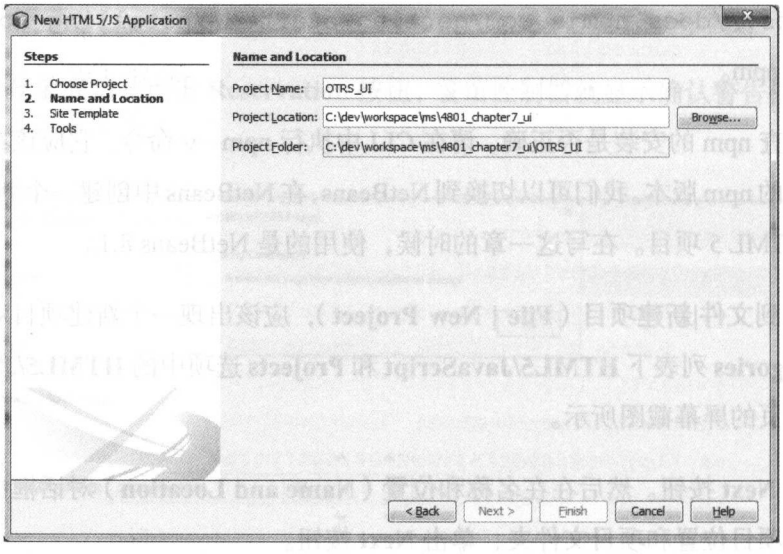
2. 要检查 npm 的安装是否正确，请在 CLI 中执行 npm -v 命令。它应该在输出中返回安装的 npm 版本。我们可以切换到 NetBeans，在 NetBeans 中创建一个新的 AngularJS JS HTML 5 项目。在写这一章的时候，使用的是 NetBeans 8.1。

3. 导航到**文件|新建项目（File | New Project）**，应该出现一个新建项目对话框。选择 **Categories** 列表下 **HTML5/JavaScript** 和 **Projects** 选项中的 **HTML5/JS Application**，如下页的屏幕截图所示。

4. 单击 **Next** 按钮。然后在在**名称和位置（Name and Location）**对话框中输入**项目名称**、**项目位置**和**项目文件夹**，单击 **Next** 按钮。

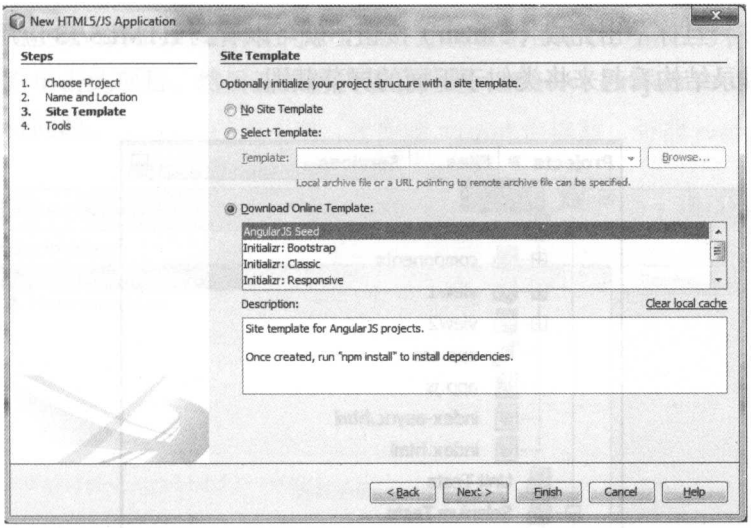


NetBeans——新建HTML5/JavaScript项目



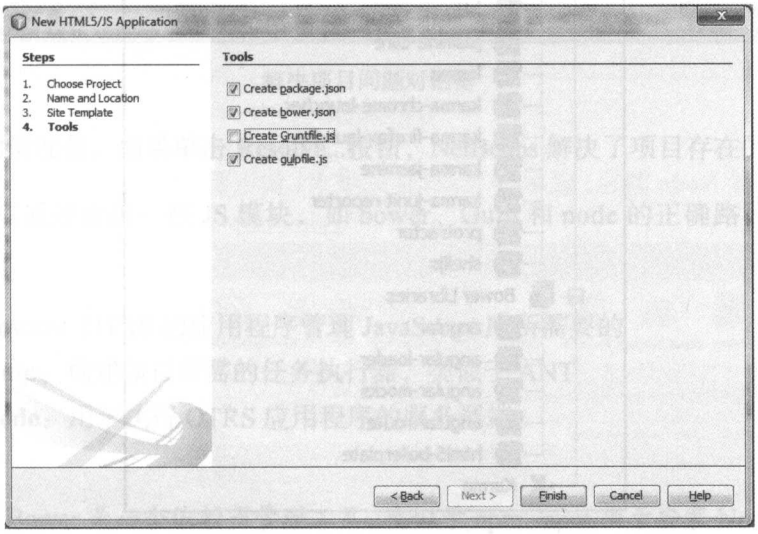
NetBeans新建项目——名称和位置

5. 在**网站模板(Site Template)**对话框中，在**下载在线模板（Download Online Template）**下选择 **AngularJS Seed** 条目：选择并单击 **Next** 按钮。AngularJS Seed 项目都可在 `https://github.com/angular/angular-seed` 获取。



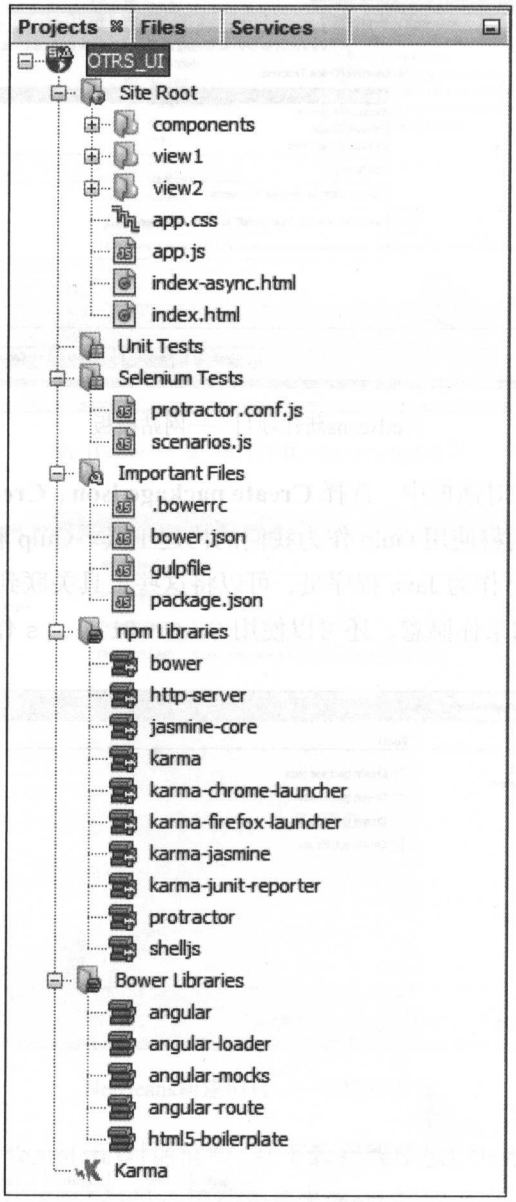
NetBeans新建项目——网站模板

6. 在工具（**Tools**）对话框中，选择 **Create package.json**、**Create bower.json** 和 **Create gulpfile.js**。我们将使用 Gulp 作为我们的构建工具。Gulp 和 Grunt 是两种最受欢迎的 JS 构建框架。作为 Java 程序员，可以将这些工具关联到 ANT。这两种工具都有各自的强项。如果你愿意，还可以使用 `Gruntfile.js` 作为构建工具。



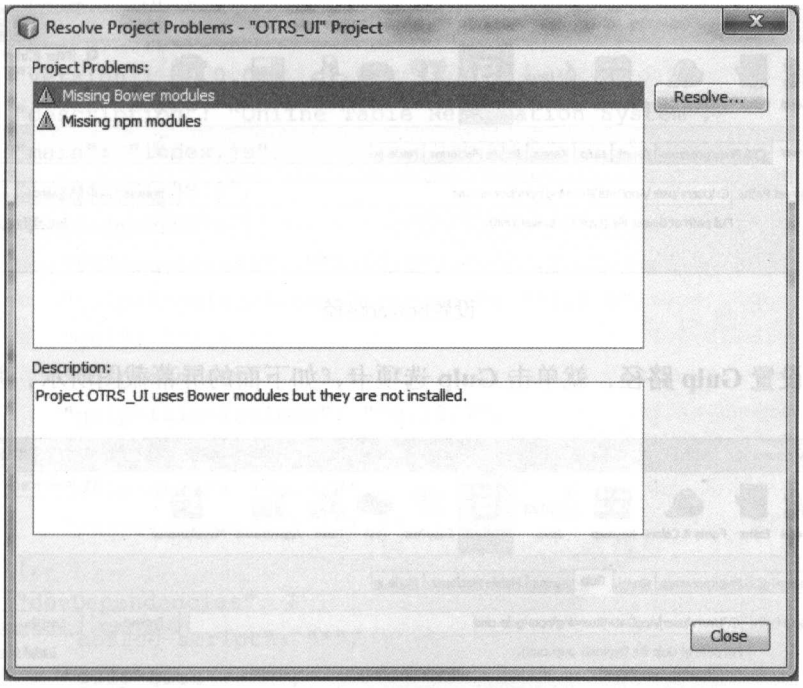
Netbeans新建项目——工具

7. 现在，一旦你单击**完成（Finish）**按钮，就可以看到 HTML5/JS 应用程序目录和文件。目录结构看起来将类似于下面的屏幕截图。



AngularJS种子目录结构

8. 如果所需的依赖项的配置有任何不正确，还将在项目中看到感叹号标记。可以使用鼠标右键单击此项目，然后选择**解决项目问题（Resolve Project Problems）**选项来解决项目问题。



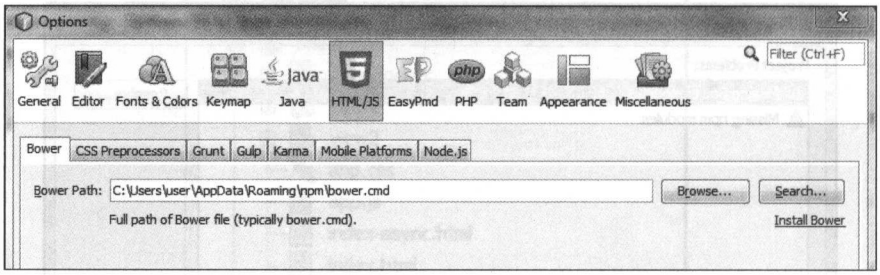
解决项目问题对话框

9. 理想的情况是，如果单击 **Resolve**...按钮，NetBeans 解决了项目存在的问题。
10. 也可以通过给出一些 JS 模块，如 bower、Gulp 和 node 的正确路径来解决一些问题：

— **Bower**：OTRS 的应用程序管理 JavaScript 库所需要的
— **Gulp**：构建项目所需的任务执行器，类似于 ANT
— **Node**：用于执行 OTRS 应用程序的服务器端

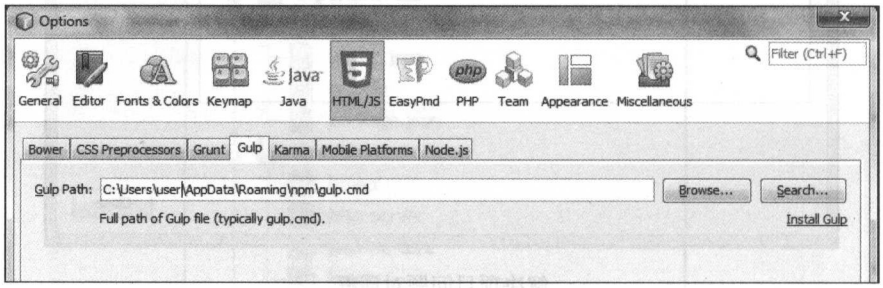
Bower 是一个依赖项管理工具，类似于 npm。npm 用于安装 Node.js 模块，而 Bower 用于管理 web 应用程序的库/组件。

11. 单击工具（**Tools**）菜单并选择选项（**Options**）。现在，在 HTML/JS 工具（顶部横栏的图标）中设置 bower、Gulp 和 node.js 路径，如下面的屏幕截图所示。要设置 bower 路径，就单击 **bower** 选项卡，如下面的屏幕截图所示，并更新路径。



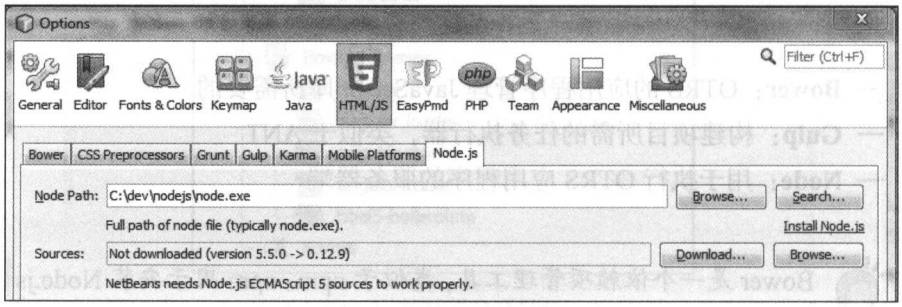
设置bower路径

12. 要设置 **Gulp 路径**，就单击 **Gulp** 选项卡，如下面的屏幕截图所示，并更新路径。



设置Gulp路径

13. 要设置 **Node 路径**，就单击 **Node.js** 选项卡，如下面的屏幕截图所示，并更新路径。



设置Node路径

14. 一旦完成这些，**package.json** 看起来将类似于下面这样。我们已了修改少数条目，如名称、描述、依赖关系等的值：

```
{
    "name": "otrs-ui",
    "private": true,
    "version": "1.0.0",
    "description": "Online Table Reservation System",
    "main": "index.js",
    "license": "MIT",
    "dependencies": {
        "coffee-script": "^1.10.0",
        "gulp-AngularJS-templatecache": "^1.8.0",
        "del": "^1.1.1",
        "gulp-connect": "^3.1.0",
        "gulp-file-include": "^0.13.7",
        "gulp-sass": "^2.2.0",
        "gulp-util": "^3.0.7",
        "run-sequence": "^1.1.5"
    },
    "devDependencies": {
        "coffee-script": "*",
        "gulp-sass": "*",
        "bower": "^1.3.1",
        "http-server": "^0.6.1",
        "jasmine-core": "^2.3.4",
        "karma": "~0.12",
        "karma-chrome-launcher": "^0.1.12",
        "karma-firefox-launcher": "^0.1.6",
        "karma-jasmine": "^0.3.5",
        "karma-junit-reporter": "^0.2.2",
        "protractor": "^2.1.0",
        "shelljs": "^0.2.6"
    },
    "scripts": {
        "postinstall": "bower install",
        "prestart": "npm install",
```

```
        "start": "http-server -a localhost -p 8000 -c-1",
        "pretest": "npm install",
        "test": "karma start karma.conf.js",
        "test-single-run": "karma start karma.conf.js  --single-run",
        "preupdate-webdriver": "npm install",
        "update-webdriver": "webdriver-manager update",
        "preprotractor": "npm run update-webdriver",
        "protractor": "protractor e2e-tests/protractor.conf.js",
        "update-index-async": "node -e \"require('shelljs/
global'); sed('-i', /\\/\\/@@NG_LOADER_START@@
[\\s\\S]*\\/\\/@@NG_LOADER_END@@/, '//@@NG_LOADER_
START@@\\n' + sed(/sourceMappingURL=AngularJS-loader.min.
js.map/,'sourceMappingURL=bower_components/AngularJS-loader/
AngularJS-loader.min.js.map','app/bower_components/AngularJS-
loader/AngularJS-loader.min.js') + '\\n//@@NG_LOADER_END@@', 'app/
index-async.html');\""
    }
}
```

15. 然后，我们会更新 bower.json，如下所示：

```
{
    "name": "OTRS-UI",
    "description": "OTRS-UI",
    "version": "0.0.1",
    "license": "MIT",
    "private": true,
    "dependencies": {
        "AngularJS": "~1.5.0",
        "AngularJS-ui-router": "~0.2.18",
        "AngularJS-mocks": "~1.5.0",
        "AngularJS-bootstrap": "~1.2.1",
        "AngularJS-touch": "~1.5.0",
        "bootstrap-sass-official": "~3.3.6",
        "AngularJS-route": "~1.5.0",
        "AngularJS-loader": "~1.5.0",
        "ngstorage": "^0.3.10",
        "AngularJS-resource": "^1.5.0",
```

```
        "html5-boilerplate": "~5.2.0"
    }
}
```

16. 下一步，我们将修改.bowerrc 文件，指定 Bower 用来存储在 bower. json 中定义的组件的目录。我们将把 bower 组件存储在 app 目录下。

```
{
  "directory": "app/bower_components"
}
```

17. 下一步，我们将设置 gulpfile.js。我们会使用 CoffeeScriptto 来定义 Gulp 任务。因此，我们在 gulpfile.js 中只定义 CoffeeScript，而实际的任务将在 gulpfile.coffee 中定义。让我们看看 gulpfile.js 的内容：

```
require('coffee-script/register');
require('./gulpfile.coffee');
```

18. 在这一步，我们将定义 gulp 配置。我们使用 CoffeeScript 来定义 gulp 文件。写在 CoffeeScript 中的 Gulp 文件名是 gulpfile.coffee。默认的任务定义为 default_sequence:

```
default_sequence = ['connect', 'build', 'watch']
```

根据定义的默认顺序任务，首先它将连接到服务器，然后构建 web 应用程序，并监控（watch）所做的更改。监控将有助于呈现我们对代码的修改，并将立即显示在 UI 上。

在此脚本中最重要的部分是 connect 和 watch，别的内容都一目了然。

- gulp-connect：这是一个 gulp 插件来运行 web 服务器，它还允许现场重新加载。
- gulp-watch：这是一个文件监控程序，使用 chokidar 并发出 vinyl 对象（描述此文件的对象——其路径和内容）。简单地说，我们可以认为，gulp-watch 监控文件的更改并触发任务。

gulpfile.coffee:

```
gulp          = require('gulp')
```

```
gutil          = require('gulp-util')
del            = require('del');
clean          = require('gulp-clean')
connect        = require('gulp-connect')
fileinclude    = require('gulp-file-include')
runSequence    = require('run-sequence')
templateCache  = require('gulp-AngularJS-templatecache')
sass           = require('gulp-sass')

paths =
  scripts:
    src: ['app/src/scripts/**/*.js']
    dest: 'public/scripts'
  scripts2:
    src: ['app/src/views/**/*.js']
    dest: 'public/scripts'
  styles:
    src: ['app/src/styles/**/*.scss']
    dest: 'public/styles'
  fonts:
    src: ['app/src/fonts/**/*']
    dest: 'public/fonts'
  images:
    src: ['app/src/images/**/*']
    dest: 'public/images'
  templates:
    src: ['app/src/views/**/*.html']
    dest: 'public/scripts'
  html:
    src: ['app/src/*.html']
    dest: 'public'
  bower:
    src: ['app/bower_components/**/*']
    dest: 'public/bower_components'

#把bower 模块复制到public目录
gulp.task 'bower', ->
```

```
  gulp.src(paths.bower.src)
  .pipe gulp.dest(paths.bower.dest)
  .pipe connect.reload()

#把scripts复制到public目录
gulp.task 'scripts', ->
  gulp.src(paths.scripts.src)
  .pipe gulp.dest(paths.scripts.dest)
  .pipe connect.reload()

#把scripts2复制到public目录
gulp.task 'scripts2', ->
  gulp.src(paths.scripts2.src)
  .pipe gulp.dest(paths.scripts2.dest)
  .pipe connect.reload()

#把styles复制到public目录
gulp.task 'styles', ->
  gulp.src(paths.styles.src)
  .pipe sass()
  .pipe gulp.dest(paths.styles.dest)
  .pipe connect.reload()

#把images复制到public目录
gulp.task 'images', ->
  gulp.src(paths.images.src)
  .pipe gulp.dest(paths.images.dest)
  .pipe connect.reload()

#把fonts复制到public目录
gulp.task 'fonts', ->
  gulp.src(paths.fonts.src)
  .pipe gulp.dest(paths.fonts.dest)
  .pipe connect.reload()

#把html复制到public目录
gulp.task 'html', ->
```

```
  gulp.src(paths.html.src)
  .pipe gulp.dest(paths.html.dest)
  .pipe connect.reload()

#在一个单独的 js 文件中编译AngularJS 模板
gulp.task 'templates', ->
  gulp.src(paths.templates.src)
  .pipe(templateCache({standalone: true}))
  .pipe(gulp.dest(paths.templates.dest))

#从公共目录删除内容
gulp.task 'clean', (callback) ->
  del ['./public/**/*'], callback;

#Gulp 连接任务，部署公共目录
gulp.task 'connect', ->
  connect.server
    root: ['./public']
    port: 1337
    livereload: true

gulp.task 'watch', ->
  gulp.watch paths.scripts.src, ['scripts']
  gulp.watch paths.scripts2.src, ['scripts2']
  gulp.watch paths.styles.src, ['styles']
  gulp.watch paths.fonts.src, ['fonts']
  gulp.watch paths.html.src, ['html']
  gulp.watch paths.images.src, ['images']
  gulp.watch paths.templates.src, ['templates']

gulp.task 'build', ['bower', 'scripts', 'scripts2', 'styles',
'fonts', 'images', 'templates', 'html']

default_sequence = ['connect', 'build', 'watch']

gulp.task 'default', default_sequence

gutil.log 'Server started and waiting for changes'
```

19. 一旦我们完成前面的更改，我们会使用以下命令安装 Gulp：

```
npm install --no-optional gulp
```

20. 此外，我们使用下面的命令安装其他 Gulp 库，如 `gulp-clean`、`gulp-connect`，等等：

```
npm install --save --no-optional gulp-util gulp-clean gulp-connect
gulp-file-include run-sequence gulp-AngularJS-templatecache gulp-
sass
```

21. 现在，我们可以使用下面的命令安装在 `bower.json` 文件中定义的 bower 依赖项：

```
bower install -save
```

```
$ bower install --save
bower angular-route#~1.4.0  not-cached git://github.com/angular/bower-angular-route.git#~1.4.0
bower angular-route#~1.4.0     resolve git://github.com/angular/bower-angular-route.git#~1.4.0
bower angular#~1.4.0        not-cached git://github.com/angular/bower-angular.git#~1.4.0
bower angular#~1.4.0           resolve git://github.com/angular/bower-angular.git#~1.4.0
bower angular-loader#~1.4.0 not-cached git://github.com/angular/bower-angular-loader.git#~1.4.0
bower angular-loader#~1.4.0    resolve git://github.com/angular/bower-angular-loader.git#~1.4.0
bower angular-mocks#~1.4.0  not-cached git://github.com/angular/bower-angular-mocks.git#~1.4.0
bower angular-mocks#~1.4.0     resolve git://github.com/angular/bower-angular-mocks.git#~1.4.0
bower html5-boilerplate#~5.2.0 not-cached git://github.com/h5bp/html5-boilerplate.git#~5.2.0
bower html5-boilerplate#~5.2.0    resolve git://github.com/h5bp/html5-boilerplate.git#~5.2.0
bower html5-boilerplate#5.2.0     download https://github.com/h5bp/html5-boilerplate/archive/5.2.0.tar.gz
bower angular#1.4.9               download https://github.com/angular/bower-angular/archive/v1.4.9.tar.gz
bower angular-loader#1.4.9        download https://github.com/angular/bower-angular-loader/archive/v1.4.9.tar.gz
bower angular-route#1.4.9         download https://github.com/angular/bower-angular-route/archive/v1.4.9.tar.gz
bower angular-mocks#1.4.9         download https://github.com/angular/bower-angular-mocks/archive/v1.4.9.tar.gz
bower angular-loader#1.4.9         extract archive.tar.gz
bower angular-loader#1.4.9        resolved git://github.com/angular/bower-angular-loader.git#1.4.9
bower html5-boilerplate#5.2.0      extract archive.tar.gz
bower angular-route#1.4.9          extract archive.tar.gz
bower angular-route#1.4.9         resolved git://github.com/angular/bower-angular-route.git#1.4.9
bower html5-boilerplate#5.2.0  invalid-meta html5-boilerplate is missing "main" entry in bower.json
bower html5-boilerplate#5.2.0  invalid-meta html5-boilerplate is missing "ignore" entry in bower.json
bower html5-boilerplate#5.2.0     resolved git://github.com/h5bp/html5-boilerplate.git#5.2.0
bower angular-mocks#1.4.9          extract archive.tar.gz
bower angular-mocks#1.4.9         resolved git://github.com/angular/bower-angular-mocks.git#1.4.9
bower angular#1.4.9               progress received 0.3MB of 0.5MB downloaded, 53%
bower angular#1.4.9               progress received 0.3MB of 0.5MB downloaded, 60%
bower angular#1.4.9               progress received 0.4MB of 0.5MB downloaded, 77%
bower angular#1.4.9               progress received 0.4MB of 0.5MB downloaded, 88%
bower angular#1.4.9               progress received 0.5MB of 0.5MB downloaded, 98%
bower angular#1.4.9                extract archive.tar.gz
bower angular#1.4.9               resolved git://github.com/angular/bower-angular.git#1.4.9
bower angular-loader#1.4.9         install angular-loader#1.4.9
bower angular-route#1.4.9          install angular-route#1.4.9
bower html5-boilerplate#5.2.0      install html5-boilerplate#5.2.0
bower angular-mocks#1.4.9          install angular-mocks#1.4.9
bower angular#1.4.9                install angular#1.4.9

angular-loader#1.4.9 app\bower_components\angular-loader
└── angular#1.4.9

angular-route#1.4.9 app\bower_components\angular-route
└── angular#1.4.9

html5-boilerplate#5.2.0 app\bower_components\html5-boilerplate

angular-mocks#1.4.9 app\bower_components\angular-mocks
└── angular#1.4.9

angular#1.4.9 app\bower_components\angular
```

示例输出——bower install --save

22. 这是安装程序的最后一步。在这里，我们将确认目录结构应该像下面这样。我们会把 src 和发布的工件(在./public 目录)作为单独的目录保留。因此，以下的目录结构不同于默认 AngularJS 种子项目：

```
+---app
|   +---bower_components
|   |   +---AngularJS
|   |   +---AngularJS-bootstrap
|   |   +---AngularJS-loader
|   |   +---AngularJS-mocks
|   |   +---AngularJS-resource
|   |   +---AngularJS-route
|   |   +---AngularJS-touch
|   |   +---AngularJS-ui-router
|   |   +---bootstrap-sass-official
|   |   +---html5-boilerplate
|   |   +---jquery
|   |   \---ngstorage
|   +---components
|   |   \---version
|   +---node_modules
|   +---public
|   |   \---css
|   \---src
|       +---scripts
|       +---styles
|       +---views
+---e2e-tests
+---nbproject
|   \---private
+---node_modules
+---public
|   +---bower_components
|   +---scripts
|   +---styles
\---test
```

一些好的阅读参考资料：

- *AngularJS by Example*，*Packt* 出版社（`https://www.packtpub.com/webdevelopment/angularjs-example`）
- *AngularSeed Project*（`https://github.com/angular/angular-seed`）
- *Angular UI*（`https://angular-ui.github.io/bootstrap/`）
- *Gulp*（`http://gulpjs.com/`）

小结

在本章中，我们学习了新的动态 web 应用程序开发。多年来，它已经完全改变了。web 应用程序前端是完全用纯 HTML 和 JavaScript 开发的，而不是使用 JSP、servlet、ASP 等任何服务器端技术开发的。使用 JavaScript UI 的应用程序开发现在有自己的开发环境，如 npm、bower 等。我们探讨了用来开发 web 应用程序的 AngularJS 框架。它通过提供对引导和处理 AJAX 调用的`$https` 服务的内置功能和支持，使得这项工作变得更容易。

希望你掌握 UI 开发概述、现代应用程序的开发，以及与服务器端微服务集成的方式。在下一章，我们将学习微服务设计的最佳做法和一般原则，将提供有关使用微服务开发的行业做法和范例的详细信息，它还将包含微服务实现在哪里出错了和如何才能避免这类问题的例子。

8

最佳做法和一般原则

在为获得微服务示例项目开发经验付出这么多辛苦的工作后，你一定会想如何避免常见的错误，并改进基于微服务的产品和服务的开发全过程。我们可以按照这些原则或准则，来简化微服务的开发过程，并避免或减少潜在的局限性。我们将在这一章着重介绍这些关键概念。

这一章分为以下三个部分：

- 概述和心态
- 最佳做法和原则
- 微服务框架和工具

概述和心态

无论对新的和现有的产品和服务，都可以实现基于微服务的设计。与从零开始开发和设计新的系统较容易，而对已经投入运行的系统做更改较难的观点相反，每种方法各有其自身的挑战和优势。

例如，由于新产品或服务没有现有的系统设计，在设计系统时有自由和灵活性，而无须对其影响给予任何考虑。但是，对于新的系统，没有明确的功能和系统需求，这些具备成熟形状需要时间。另一方面，对于成熟的产品和服务，对功能和系统需求有详细的知识和信息。不过，会遇到减轻设计变化的影响带来的风险的挑战。因此，在把整体式的生产系统更新到微服务的时候，比起从零开始构建一个系统，将需要更好的规划。

有经验的成功软件设计专家和架构师总是评估利弊，并对现存的系统做出任何改变采取审慎的态度。人们不应该因为可能会很酷或时尚而对现有系统设计进行更改。因此，如果想要把现有生产系统的设计更新为微服务，需要在做这个决定之前评估所有的利弊。

我相信整体系统为升级到一个成功的基于微服务的设计提供了很好的平台。很显然，在这里我们不讨论成本。现有系统和功能的足够知识，使你能够基于现有系统的功能和这些功能之间的交互方式对其进行划分，并在此基础上建立微服务。此外，如果整体式产品已经是以某种方式模块化的，那么通过公开的 API，而不是**应用程序二进制接口（Application Binary Interface，ABI）**直接转化微服务可能是最简单的实现微服务架构的方法。一个基于微服务的系统的成功更依赖于微服务及其交互协议而不是别的。

话虽如此，这并不意味着如果你从零开始就不能开发出一个成功的基于微服务的系统。然而，建议以整体设计为基础开始一个新的项目，它会为你提供功能与系统的角度和理解。它使你能够快速找到瓶颈，并指导你找出任何可以使用微服务开发的潜在功能。在这里，我们还没有讨论项目的大小，这是另一个重要因素。我们将在下一节讨论这个主题。

在当今的云时代和敏捷开发世界中，从做出任何改变到变化生效只需要用一个小时的时间。在今天的竞争环境中，每个组织都想具有迅速地向用户提供功能的优势。持续开发、集成和部署都是生产交付过程这个完全自动化过程的一部分。

如果你提供基于云环境的产品或服务，这么做就更有意义。然后，基于微服务的系统使团队能够灵活应对来解决任何问题，或向用户提供一个新的功能。

因此，在做出从头开始一个新的基于微服务的项目或计划把现有的整体系统的设计升级到一个基于微服务的系统的决定之前，需要评估所有利弊。必须倾听和理解团队中存在的不同想法和观点，并且需要采取审慎的态度。

最后，我想要分享具有一个更好的流程和高效的系统对于生产系统成功的重要性。在当今的时代，有一个基于微服务的系统并不能保证生产系统的成功，而独立应用程序并不意味着你不能有一个成功的生产系统。Netflix 这个基于微服务的云视频租赁服务和 Etsy 这个整体式电子商务平台，这两者都是成功的实际生产系统的例子（参见本章后面参考资料一节中一个有趣的 Twitter 讨论链接）。因此，流程和敏捷性也是生产系统成功的关键。

最佳做法和原则

正如我们从第一章学到的，微服务是实现**面向服务的架构（Service Oriented Architecture，SOA）**的轻量级的风格。最重要的是，微服务的定义是不严格的，这使你可以按想要的方式和根据需要灵活地开发微服务。同时，需要确保遵循几条标准的做法和原则，以使你的工作更容易一些，并成功实施基于微服务的架构。

Nanoservice（不推荐）、规模和整体性

在你的项目中，每个微服务都应体积微小并执行一种功能或特性（例如，用户管理），且足够独立地自主执行功能。

以下两条引文来自 Mike Gancarz(X windows 系统的设计成员），它定义了 UNIX 哲学的首要戒律之一，它也适合微服务范式：

> “小即是美。”
>
> “让每个程序做好一件事。”

那么，在当今时代，当你有一个能减少**代码行（lines of code，LOC）**的框架（例如 Finangle）时，如何定义规模呢？此外，许多现代编程语言，如 Python 和 Erlang，都不那么烦琐。这使得是否要使此代码成为微服务变得难以决定。

显然，你可能会实现一个LOC很少的微服务，但它实际上不是微服务，而是nanoservice。

Arnon Rotem-Gal-Oz 对 nanoservice 的定义如下：

> “Nanoservice 是一种反模式，它的服务的粒度太细。Nanoservice 是一个（通信、维护，等等）开销超过其用途的服务。”

因此，基于功能来设计微服务很有意义。领域驱动设计使得在领域级别定义功能变得很容易。

如前文所述，项目规模是决定是否为项目实施微服务，或确定你想要的微服务数量时的一个关键因素。在一个简单的小型项目中，使用整体式架构是有意义的。例如，基于我们在**第 3 章**中学到的领域设计，你会清楚地了解功能需求，了解可用来绘制边界之间各种

功能或特性的事实。例如，在我们已实现的示例项目(OTRS)中，假如你不想要把 API 公开给客户，或不想把它作为 SaaS 使用，或者做决定之前有很多你想要评估的相似参数，使用整体设计来开发同一项目是很容易的。

可以在以后需要时把整体项目迁移到微服务设计。因此，至关重要的是，应该用模块化方式开发整体项目并在每个层之间都具有松耦合，并确保不同的功能和特性之间有预定义的接触点和边界。此外，还应相应地设计你的数据源，如数据库。即使你不打算迁移到一个基于微服务的系统，它也将使 bug 修复和增强易于实现。

当你迁移到微服务时，注意上述各点将减轻你可能会遇到的任何困难。

一般来说，如前几章所述，开发大型或复杂的项目应该使用基于微服务的架构，因为它提供了许多优点。

尽管我建议把初始项目开发为整体式的，但一旦你更好地了解了项目功能和项目的复杂性，那么你就可以将其迁移到微服务。理想情况下，一个已开发出的初始原型应该给你提供功能的边界，使你能够做出正确的选择。

持续集成和部署

必须具备一个持续的集成和部署过程，它给你更快提供更改和及早发现 bug 的好处。因此，每个服务都应该有其自身的集成和部署过程。此外，它必须被自动化。这个任务有很多工具可用，例如 Teamcity、Jenkins，等等，它们都得到广泛使用。它可以帮助你自动生成过程——及早发现构建的失败，尤其是当你将更改集成到主线时。

也可以将测试集成到每个自动的集成和部署过程中。**集成测试（Integration Testing）**负责测试系统不同部分的交互，如两个接口（API 提供者和使用者）之间，或系统不同的组件或模块，如 DAO 与数据库之间，等等。集成测试是重要的，因为它测试模块之间的接口。单个模块首先进行隔离测试，然后集成测试，以检查联合的行为并验证需求被正确实现。因此，在微服务中，集成测试是一个验证 API 的关键工具。在下一节，我们将介绍更多关于它的内容。

最后，在此过程在其中部署构建完成的软件的 DIT 计算机上，可以看到更新主线发生更改。

这个过程不会在这里结束，可以制作一个容器，如 docker，并把它交给 WebOps 团队，或拥有一个单独的进程，将它传递到已配置的位置，或将它部署到 WebOps 阶段性环境。从这里，一旦由指定的主管部门批准，它就可以被直接部署到生产系统上。

系统/端到端测试自动化

测试是交付任何产品和服务的重要部分。你不想要交付给客户的应用程序有很多错误。过去，在瀑布模型很流行的时候，一个组织经常在交付给客户之前采取一至六个月或更长时间的测试阶段。近年来，在敏捷过程变得流行后，自动化得到更多重视。类似于前面的测试，自动化也是强制性的。

无论你是否遵循**测试驱动开发(Test Driven Development，TDD)**，我们都必须具备系统或端到端的自动化测试。对于测试你的业务场景，它是非常重要的，对于可能开始从你的 REST 调用数据库检查，或从 UI 应用程序到数据库检查的端到端测试，也是如此。

此外，如果你有公共 API，测试 API 也是很重要的。

这样做会确保任何更改都不会破坏任何功能，并确保交付无缝、无缺陷的产品。如上节所述，每个模块都使用单元测试进行单独测试，以检查一切都在按预期运行，然后，无论实现得正确与否，集成测试都在不同的模块之间执行，以检查预期的联合行为并校验需求。在集成测试后，还会执行验证功能和功能需求的功能测试。

所以，如果单元测试确保单个模块独立工作情况良好，集成测试确保不同模块之间的交互均按预期方式工作。如果单元测试工作情况良好，它意味着集成测试失败的概率大大降低。同样，集成测试确保功能测试很有可能获得成功。

据推测，人们始终会保持所有类型的测试更新，无论这些是单元级测试还是端到端测试场景。

自我监控和记录

微服务应当提供服务本身的信息和它所依赖的各种资源的状态。服务信息包括处理一个请求的平均、最小和最大时间，成功和失败的请求的数量，能够跟踪请求、内存使用情况等的统计信息。

Adrian Cockcroft 在 2015 年的 **Glue 会议（Glue Conference，Glue Con）**上强调的下面几个做法对于监控微服务是非常重要的。它们中的大部分对任何监控系统都有效：

- 分析指标含义的代码，要比收集、移动、存储和显示度量指标的代码花更多时间来开发。

  这不仅有助于提高生产效率，而且还提供了重要参数来微调微服务和提高系统效率。这个观点是开发更多的分析工具，而不是开发更多的监控工具。
- 用来显示延迟的指标需要小于人类的注意力跨度。根据 Adrian 的说法，这意味着少于 10 秒。
- 验证你的测量系统具有足够的准确度与精度。收集响应时间的直方图。
- 准确的数据能使决策更快，并允许你微调直到所需的精度水平。他还建议最好用的显示响应时间的图形是直方图。
- 监控系统需要比被监控的系统有更高的可用性和可扩展性。
- 总而言之，你不能依靠本身并不稳定或 24/7 可用的系统。
- 针对分布式、短暂性、云本机、容器化的微服务进行优化。
- 用模型来拟合指标以理解相互关系。

监控是微服务架构的关键组成部分。根据项目的大小，你可能有十几个到数以千计的微服务（对于大企业的大项目）。即使对于扩展性和高可用性，组织为每个微服务创建集群或负载均衡池/仓，甚至基于每个版本的微服务创建单独的池。最终，它增加了你需要监控的资源，包括微服务的每个实例。此外，重要的一点是，你应该具备一个进程，每当有什么闪失，你就立即知道它，或更好的是，在出了什么差错前提前收到一个警告通知。因此，有效且高效的监控对构建和使用微服务架构至关重要。Netflix 使用情况安全监控使用诸如 Netflix Atlas（实时业务监控，处理 12 亿指标）、Security Monkey（用于监控基于 AWS 的环境的安全）、Scumblr（情报收集工具）和 FIDO（对事件的分析和自动化事件报告）的工具。

日志记录是微服务不应被忽视的另一个重要方面。具有有效的日志记录关系重大。因为可能有 10 个或更多的微服务，所以管理日志记录是一项艰巨的任务。

对于我们的示例项目，我们用 MDC 日志记录，在某种程度上，对单个微服务日志记录，这就足够了。然而，我们还需要对一个完整的系统的日志记录或集中式日志记录。我们还需要日志的汇总统计数据。做这项工作的工具，包括 Loggly 和 Logspout。

请求和生成的相关事件提供了请求的总体视角。为追踪任何事件和请求，分别用服务 ID 和请求 ID 来关联事件和请求是至关重要的。也可以将事件的内容，例如消息、严重程度、类名等关联到服务 ID。

每个微服务都使用独立的数据存储区

如果你还记得，你可以发现有关微服务的最重要特征是，微服务与其他微服务隔离的运行方式，通常是作为独立应用程序运行的。

遵守这项规则，建议你不要跨多个微服务使用相同的数据库或任何其他数据存储区。在大型项目中，你可能会有不同的团队在同一个项目中工作，并且你想要能够灵活地为每个微服务选择最适合此微服务的数据库。

现在，这也带来了一些挑战。

例如，对可能为同一项目中的不同微服务工作的团队，如果此项目共享相同的数据库结构，会产生以下问题：存在一个微服务的变化可能会影响其他微服务模型的可能性。在这种情况下，一个微服务的变化可能影响依赖于它的微服务，所以你也需要改变依赖模型的结构。

要解决此问题，微服务应该基于 API 驱动的平台开发。每个微服务都将公开其 API，它们可以由其他微服务使用。因此，你也需要开发 API，这是集成不同的微服务的需要。

同样，由于有不同的数据存储区，实际项目数据也分散在多个数据存储区，这使数据管理更复杂，因为单独的存储系统更容易变得不同步或变得不一致，并且外键会发生意外的变化。要解决这种问题，你需要使用**主数据管理（Master Data Management，MDM）**工具。MDM 工具在后台运行，并且如果它发现任何不一致，就修复它们。对于 OTRS 示例，它可能会检查存储预订请求 ID 的每个数据库，以验证相同的 ID 存在于所有的数据库中（换句话说，在任何一个数据库中都不存在任何丢失的 ID 或额外的 ID）。市场上的 MDM 工具包括 Informatica、IBM MDM 高级版、Oracle Siebel UCM、Postgres（主数据流复制）、mariadb（主/主配置），等等。

如果没有满足你的要求的现有产品，或者你对任何专有的产品都不感兴趣，那么你可

以编写你自己的 MDM 工具。目前，API 驱动的开发和平台减少这些问题的复杂性，因此，至关重要的是，微服务应在 API 平台上开发。

事务边界

我们已经在**第 3 章**学过了领域驱动设计的概念。如果你还没有彻底掌握这些概念，请复习它们，因为这使你垂直地理解状态。因为我们专注于基于微服务的设计，结果是，我们有一个系统的系统，其中每个微服务都表示一个系统。在这种环境中，找到任何给定时刻的整个系统的状态是非常具有挑战性的。如果你熟悉分布式应用程序，可能会很好地适应在这种环境中的状态。

具有描述在任何给定时间哪个微服务拥有一条消息的事务边界是非常重要的。你需要能参与事务的方式或者过程、事务化的路由和错误处理程序、等效使用者和补偿操作。确保跨异构系统的事务性行为不是件容易的事，但有工具可为你做这个工作。

例如，Camel 有很强的事务性功能，可帮助开发人员轻松地创建具有事务性行为的服务。

微服务框架和工具

不去重新发明轮子总是更好的。因此，我们想探讨都有哪些已经可用的工具，能提供使微服务的开发和部署更容易的平台、框架和功能。

在本书中，我们已经广泛地使用了 Spring Cloud，由于同样的原因，它提供使微服务非常容易地开发所需的所有工具和平台。Spring Cloud 使用 Netflix **开放源码软件（OSS）**。让我们探讨 Netflix OSS——一个完整的软件包。

我还补充了每个工具将如何有助于建立良好的微服务架构的简要概述。

Netflix 开放源码软件（OSS）

Netflix 开放源码软件中心是基于 Java 的微服务开放源码项目最流行和最广泛使用的开放源码软件。世界上最成功的视频租赁服务依赖于它。Netflix 已经有超过 4 000 万用户，他们在全球各地使用其服务。Netflix 是一个纯粹的基于云平台的解决方案，在微服务架构

的基础上开发。可以说，每当有人谈到微服务时，Netflix 都是进入你脑海的第一个名字。让我们讨论它提供的各种工具。在开发示例 OTRS 应用程序时，我们已经讨论了其中的很多工具。然而，有几个工具我们还未探讨过。在这里，我们将只对每个工具进行概述，而不是详细讲解。这将给你带来微服务架构的实际特点和它在云平台中使用的总体思路。

构建——Nebula

Netflix Nebula 是一种使你更容易使用 Gradle（类似 Maven 的构建工具）来生成微服务的 Gradle 插件集合。对于我们的示例项目，由于我们已使用了 Maven，因此我们没有机会在本书中详细探讨 Nebula。然而，研究它会很有趣。对于开发人员来说，最重要的 Nebula 功能是消除 Gradle 生成文件中的许多样板代码，这使得开发人员能够把重点放在编码上面。

有一个很好的构建环境，尤其是 CI/CD（持续集成和持续部署）是微服务开发和与敏捷开发保持一致必备的。Netflix Nebula 使你的构建过程更轻松、更高效。

部署和交付——Spinnaker 与 Aminator

一旦你生成的软件已准备就绪，你会想要将此软件移动到**亚马逊网络服务（Amazon Web Services，AWS）**EC2 中。Aminator 使用**亚马逊机器映像（Amazon Machine Image，AMI）**的形式来创建生成的软件并将其打包成映像文件。Spinnaker 然后将这些 AMI 部署到 AWS。

Spinnaker 是高速并高效地发布代码更改的持续交付平台。Spinnaker 还支持其他云服务，例如 Google Computer Engine 和 Cloud Foundry。

你想要将最新的微服务软件部署于类似 EC2 的云环境中，Spinnaker 和 Aminator 可以帮助你自动地完成这件事。

服务注册和发现——Eureka

正如我们已在本书中探讨的，Eureka 提供了负责微服务注册和发现的服务。最重要的是，Eureka 也用于中间层（承载不同的微服务的进程）负载均衡。Netflix 也使用 Eureka 以

及其他工具，像 Cassandra 或 memcached，以提高其整体可用性。

服务注册和发现是微服务架构所必备的。Eureka 的用途就是这个。请参阅**第 4 章**获取有关 Eureka 的详细信息。

服务沟通——Ribbon

如果没有进程间或服务间的通信，微服务架构就没有用。功能区应用程序提供该功能。Ribbon 与 Eureka 结合实现负载均衡，与 Hystrix 结合实现容错或电路断路器操作。

除 HTTP 之外，Ribbon 还支持 TCP 和 UDP 协议。它对这些协议同时提供了异步和反应式模型的支持，它还提供缓存和批处理的功能。

因为你的项目中将会有很多微服务，你需要一种使用进程间或服务间通信的方法来处理信息。Netflix 公司为此提供了 Ribbon 工具。

电路断路器——Hystrix

Hystrix 工具用来执行电路断路器操作，也就是，容忍延迟和容错。因此，Hystrix 会停止连锁故障。Hystrix 执行实时的服务监控和属性更改操作，并支持并发。

电路断路器或容错，是用于任何项目，包括微服务的一个重要概念。一个微服务的故障不应停止你的整个系统；Netflix Hystrix 的任务是防止这一点，并在出故障时，向用户提供有意义的信息。

边缘（代理）服务器——Zuul

Zuul 是边缘服务器或代理服务器，它用来为外部应用程序如 UI 客户端、Android/iOS 应用程序或任何产品或服务提供的第三方使用者的 API 发出的请求提供服务。从概念上讲，它是一扇面向外部应用程序的门。

Zuul 允许动态路由和监控请求。它还执行安全操作，如身份验证。可以确定每个资源的身份验证要求，并拒绝任何不能满足这些要求的请求。

你需要为微服务提供边缘服务器或 API 网关。Netflix Zuul 提供此功能，请参阅**第 5 章**获取详细信息。

业务监控——Atlas

Atlas 是一个业务监控工具，它提供了接近实时的高维时间序列数据的信息。它捕获业务智能，提供目前在系统内发生的情况的概况。它提供内存中的数据存储，这使它能非常快地收集和报告大量的指标。目前，它为 Netflix 处理 13 亿个指标。

Atlas 是一个可扩展的工具。这就是为什么它从几年前处理 100 万个指标到现在可以处理 13 亿个指标的原因。Atlas 系统不仅提供了读取数据方面的可扩展性，还将它作为一部分集成到图形请求中。

Atlas 使用 Netflix Spectato 库记录高维时间序列数据。

一旦你在云环境中部署微服务，你就需要有一个监控系统来跟踪和监控所有微服务。Netflix Atlas 为你做这份工作。

可靠性监控服务——Simian Army

在云环境中，没有单个的组件可以保证 100%正常运行时间。因此，成功的微服务架构要求，使整个系统在单个云组件出现故障的情况下可用。Netflix 公司开发了一个叫 Simian Army（猿猴军队）的工具，以避免系统故障。Simian Army 保证云环境的安全、可靠和高可用性。为了实现高可用性和安全性，它使用各种服务（猴子）在云中产生各类故障，检测异常情况并测试云在这些挑战下的生存能力。它使用取自 Netflix 博客的以下服务（猴子）：

- **混沌猴子（Chaos Monkey）**：混沌猴子是一种服务，它确定系统组并随机终止某个组中的一个系统。这个服务以受控制的时间和时间间隔运作。混沌猴子只在营业时间运行，它希望工程师将收到警报并能够做出反应。
- **看守猴子（Janitor Monkey）**：看守猴子是一种在 AWS 云中运行的服务，它寻找可以清理的未使用的资源，它可以扩展用于其他云提供商和云资源。服务的时间表是可配置的。看守猴子通过对某个资源应用一组规则，确定它是否应当成为被清理

的候选者。如果任何规则确定某资源是被清理的候选者，看守猴子就对此资源做标记，并安排时间去清理它。特殊情况下，当你想要把未使用的资源保留更长的时间，在看守猴子删除资源前，资源的所有者将在清理时间前几天收到通知，天数是可配置的。

- **符合猴子（Conformity Monkey）**：是一种在 AWS 云中运行的服务，它寻找不符合最佳做法的预定义规则的实例。它可以扩展来用于其他云提供商和云资源。这个服务的时间表是可配置的。

  如果确定该实例不符合任何一条规则，猴子就向实例的所有者发送电子邮件通知。可能在有的例外情况下，对于某些应用程序要忽略关于符合特定规则的警告。
- **安全猴子（Security Monkey）**：安全猴子监控策略的更改并对某个 AWS 账户上没有安全感的配置进行提醒。安全猴子的主要目的是保证安全性，但它也是用于跟踪潜在问题的有用工具，因为它本质上是一个更改跟踪系统。
- 成功的微服务架构可以确保你的系统始终是运行的，并且单个云组件失败不会停止整个系统。Simian Army 使用许多服务来实现高可用性。

AWS 资源监控——Edda

在云环境中，没有什么是静态的。例如，虚拟宿主机实例经常发生变化，通常情况下，IP 地址可以由各种应用程序重复使用，防火墙或相关的变化也可能发生。

Edda 是跟踪这些动态的 AWS 资源的服务。Netflix 将其命名为 Edda（即北欧神话故事），它记录云管理和部署的故事。Edda 使用 AWS API 轮询 AWS 资源并记录结果。这些记录允许搜索和查看云已经随着时间的推移发生了哪些变化。例如，如果任何 API 服务器的主机正在造成任何问题，你需要找出此主机是什么，哪支团队要为它负责。

它提供了这些功能：

- **动态查询**：Edda 提供 REST API，并且它支持矩阵参数并提供让你仅检索所需的数据的字段选择器。
- **历史的变化**：Edda 维护所有 AWS 资源的历史记录。此信息可帮助你分析资源中断的原因和影响。Edda 还可以提供有关资源的当前和历史信息的不同视图。在撰写本文时，它在 MongoDB 中存储信息。

- **配置**：Edda 支持多个配置选项。一般情况下，可以从多个账户和多个区域轮询信息，还可以使用账户的组合和这些账户指向的区域。同样的，它提供 AWS、Crawler、Elector 和 MongoDB 的不同配置。
- 如果采用 AWS 来承载基于微服务的产品，那么 Edda 可用于对 AWS 资源进行监控。

主机性能监控——Vector

Vector 是一个静态的 web 应用程序，在 web 浏览器内运行。它可以用来监控安装了 Performance Co-Pilot(PCP)的主机的性能。Vector 支持 PCP 3.10 及以上版本。PCP 收集各种指标并提供给 Vector。

它根据需要提供高分辨率的正确指标。这可以帮助工程师了解系统的行为和正确地解决性能问题。

可帮助你监控远程主机的性能监控工具。

分布式配置管理——Archaius

Archaius 是一个分布式的配置管理工具，它允许你执行以下操作：

- 使用动态和类型化的属性。
- 执行线程安全的配置操作。
- 使用轮询框架检查属性更改。
- 在有序的层次结构的配置中使用回调机制。
- 使用 JConsole 检查属性并对其执行操作，因为 Archaius 提供了 JMX MBean。
- 当你有一个基于微服务的产品时，需要有一个良好的配置管理工具。Archaius 可以帮助在一个分布式的环境中配置不同类型的属性。

Apache Mesos 调度器——Fenzo

Fenzo 是用 Java 编写的用于 Apache Mesos 框架的一个调度程序库。Apache Mesos 框架查找匹配的资源，并将其分配到挂起的任务上。其主要特点如下：

- 支持长时间运行的服务风格的任务和批处理。
- 可以基于资源需求自动缩放执行主机集群。
- 支持插件，可以基于需求创建它们。
- 可以监控资源分配的故障，允许调试故障根源。

成本和云利用率——Ice

Ice 从成本和使用的角度提供云资源的全景图。它提供调配云资源分配到不同团队的最新信息，为云计算资源的最优利用增加价值。

Ice 是一个圣杯项目。用户与 Ice UI 组件交互，后者显示通过 Ice 阅读器组件发送的信息。阅读器从 Ice 处理器组件所生成的数据中提取信息。Ice 处理器组件从详细的云计费文件中读取数据信息，并将它转换成 Ice 阅读器组件可读的数据。

其他安全工具——Scumblr 和 FIDO

除了 Security Monkey，Netflix 开放源码软件也使用 Scumblr 和完全集成的防御操作（**Fully Integrated Defense Operation，FIDO**）工具。

为了跟踪你的微服务，并保护它不受经常的威胁和攻击，你需要以自动化的方式来对你的微服务进行保护和监控。Netflix Scumblr 和 FIDO 为你做这份工作。

Scumblr

Scumblr 是一个基于 Ruby on Rails 的 web 应用程序，它允许你执行定期搜索并对识别的结果执行存储/采取行动。基本上，它会利用全互联网有针对性的搜索来收集情报，从而揭露特定安全问题用于调查。

Scumblr 利用可以流程化的宝贵信息，允许对不同类型的结果设置灵活的工作流。Scumblr 利用称为 **Search Providers**（搜索提供程序）的插件进行搜索，它会检查类似以下的异常。因为它是可扩展的，可以根据需要添加任意多的检查项目：

- 泄露的凭据

- 黑客漏洞/讨论
- 攻击讨论
- 社交媒体上的安全相关讨论

完全集成的防御操作（FIDO）

FIDO 是一种安全业务流程框架，用于分析事件和自动化事件响应。它通过评价、评估和应对恶意软件来使事件的响应过程变得自动化。FIDO 的主要目的是处理评估来自当今安全栈的威胁和它们所生成的大量警报所需要的大量手动工作。

作为业务流程平台，FIDO 通过大幅减少检测、通知和应对网络攻击所需要的手动工作，可以更高效、更准确地使用现有的安全工具。有关详细信息，可以参考下面的链接：

```
https://github.com/Netflix/Fido https://github.com/Netflix
```

参考资料

- 关于整体式(Etsy)与微服务(Netflix)的比较的 Twitter 讨论， https://twitter.com/adrianco/status/441169921863860225
- *Monitoring Microservice and Containers Presentation*（监控微服务和容器演示文稿）Adrian Cockcrof 编：http://www.slideshare.net/adriancockcroft/gluecon-monitorin-gmicroservices-and-containers-a-challenge
- Nanoservice 反模式：http://arnon.me/2014/03/microservices-nanoservices/
- 微服务的架构的 Apache Camel：https://www.javacodegeeks.com/2014/09/apache-camel-for-micro%C2%ADservicearchitectures.html
- Teamcity：https://www.jetbrains.com/teamcity/
- Jenkins：https://jenkins-ci.org/
- Loggly：https://www.loggly.com/

小结

在这一章，我们探讨了最适合基于微服务的产品和服务的各种做法和原则。微服务架构是云环境的结果，它被广泛用于与基于整体式系统的对比。我们发现了少量与规模、敏捷性和测试有关的原则，它们必须具备才能成功实现微服务。

我们也概述了由 Netflix OSS 使用的不同工具，它们用于成功实现微服务架构产品和服务所需的各种关键功能。Netflix 成功地使用同样的工具提供视频租赁服务。

在下一章，读者可能会遇到各种问题，他们可能会被这些问题困住。下一章解释了在开发微服务的过程中所遇到的常见问题和它们的解决办法。

9

故障排除指南

到目前为止，我们已经学了这么多东西，我敢肯定你享受这个具有挑战性的快乐学习旅程的每时每刻。学完这一章后，我不愿意说这本书结束了，而宁愿说你正在完成第一个里程碑。跨过这个里程碑，我们就可以继续学习基于微服务的新设计范式并在云环境中实现它。我想重申，集成测试是测试微服务和 API 之间交互的重要途径。在你完成**在线餐馆订座系统（OTRS）**示例应用程序的过程中，我确信你面临许多挑战，尤其是在调试应用程序时。在这里，我们将介绍几种做法和工具，帮助你解决部署应用程序、Docker 容器和宿主机的故障。

本章包括以下三个主题：

- 日志记录和 ELK 环境
- 使用相关 ID 来进行服务调用
- 依赖项和版本

日志记录和 ELK 环境

你能想象在没有看到日志的情况下在生产系统上调试任何问题吗？简单地说，不行，因为时间是很难回去的。因此，我们需要日志记录。如果它们是按这种方式设计和编码的，日志也给我们提供有关系统的警告信号。日志记录和日志分析，对排除任何问题的故障和吞吐量、容量和系统的健康状况监控，都是一个重要的步骤。因此，有一种很好的日志平台和战略将使调试变得高效。日志是在软件开发最初的几天中最重要的关键部件之一。

微服务一般使用映像的容器，如 Docker 来部署，它们提供日志，并通过命令来帮助你读取部署在容器内的服务日志。Docker 和 Docker Compose 提供把容器内运行的、在所有容器中的服务日志分别输出的命令。请参阅以下 Docker 和 Docker Compose 的 logs 命令：

Docker logs 命令：

用法： docker logs[OPTIONS] < CONTAINER NAME >

获取容器的日志：

```
-f, --follow    跟踪日志输出
--help      打印用法
--since=""                  显示时间戳以来的日志
-t, --timestamps            显示时间戳
timestamp
--tail="all"                要从日志的末尾显示的行数
```

Docker Compose logs 日志命令：

用法： docker-compose logs [OPTIONS] [SERVICE...]

OPTIONS：

```
--no-color  产生单色输出
-f, --follow  跟踪日志输出
-t, --timestamps  显示时间戳
--tail    要从每个容器日志的末尾显示的行数
[SERVICE…]  表示容器的服务——可以给出多个
```

这些命令帮助你检查微服务和其他容器内运行的进程的日志。正如你所看到的，当你有大量的服务时，使用上面的命令将具有很大的挑战性。例如，如果你有数十个或数百个微服务，将很难跟踪每个微服务的日志。同样，你可以想象，即使没有容器，监控单独的日志记录将是多么困难。因此，你可以想象，检查和关联数十到数百个容器的日志的难度有多大。它非常耗时，并带来很少的价值。

因此，我们将使用一种类似 ELK 环境的日志聚合器和可视化工具来辅助，它将用于集中日志记录。下一节，我们将探讨这个工具。

简要概述

Elasticsearch、**Logstash**、**Kibana**（**ELK**）环境是一系列执行日志聚合、分析、可视化和监控的工具。ELK 环境提供完整日志记录平台，用来分析、可视化和监控所有日志，包括所有类型的产品日志和系统日志。如果你已经了解 ELK 环境，请跳过下一节。在这里，我们会提供 ELK 环境中的每个工具的简介。

Elasticsearch

Elasticsearch 是最受欢迎的企业全文搜索引擎之一。它是开放源码的软件，它是可分布式的，并支持多租户。一个单独的 Elasticsearch 服务器存储多个索引（每个索引都表示一个数据库），并且单个查询可以搜索多个索引的数据。它是一个分布式的搜索引擎并支持聚类。

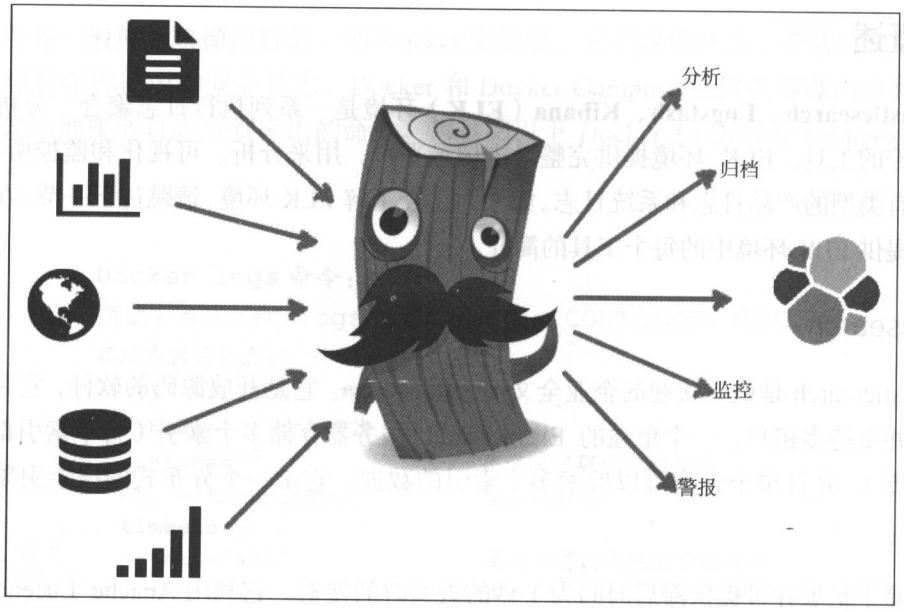
它易于扩展并可提供滞后时间为 1 秒的近实时的搜索。它利用 Apache Lucene 和 Java 开发。Apache Lucene 也是免费的，开放源码，它提供了 Elasticsearch 的核心，又名信息检索软件库。

Elasticsearch API 本质上广泛且非常精细。Elasticsearch 提供了一个基于 JSON 的架构，更少的存储，并用 JSON 表示数据模型。Elasticsearch API 使用 JSON 文档处理 HTTP 请求和响应。

Logstash

Logstash 是具有实时管道能力的开放源码信息收集引擎。简单地说，它收集、分析、处理并存储数据。由于 Logstash 具有数据管道功能，它能帮助你处理任何来自各种系统的事件数据，如日志。Logstash 作为代理来运行，它收集数据，解析、过滤它们，并将其输出发送到一个指定的应用程序，如 Elasticsearch 或在控制台上的简单的标准输出。

它也有一个很好的插件生态系统（图像来自 `www.elastic.co`）：



Logstash生态系统

Kibana

Kibana 是一个开源分析和可视化的 web 应用程序，它被设计为与 Elasticsearch 配合工作。Kibana 用于搜索、查看和与存储在 Elasticsearch 索引中的数据进行交互。

它是一个基于浏览器的 web 应用程序，可以用各种图表、表和地图执行高级的数据分析和可视化数据。此外，它是一个零配置的应用程序。因此，安装它以后不需要任何编码或额外的基础设施。

ELK 环境安装

一般来说，这些工具都是单独安装的，然后配置为相互通信。这些组件的安装是相当简单的。从指定的位置下载可安装的工件，并按照下一节所示的步骤安装。

下面提供的安装步骤是设置你想要运行的 ELK 环境所需的基本安装程序的一部分。由于此安装在我的本地主机上完成，我已用了主机 localhost。它可以容易地改为任何你想要的单独的主机名。

安装 Elasticsearch

我们可以按照如下步骤来安装 Elasticsearchby：

1. 从 https://www.elastic.co/downloads/elasticsearch 下载最新的 Elasticsearch 软件。
2. 将其解压缩到你的系统中的所需位置。
3. 确保安装了最新的 Java 版本和设置了 JAVA_HOME 环境变量。
4. 转到 Elasticsearch 主目录，在基于 Unix 的系统上，运行 bin/elasticsearchon，而在 Windows 上，运行 bin/elasticsearch.bat。
5. 打开任何浏览器，并输入 http://localhost:9200/。成功安装后应该会提供一个类似于如下所示的 JSON 对象：

```
{
  "name" : "Leech",
  "cluster_name" : "elasticsearch",
  "version" : {
    "number" : "2.3.1",
    "build_hash" : "bd980929010aef404e7cb0843e61d0665269fc39",
    "build_timestamp" : "2016-04-04T12:25:05Z",
    "build_snapshot" : false,
    "lucene_version" : "5.5.0"
  },
  "tagline" : "You Know, for Search"
}
```

默认情况下，不安装 GUI。可以通过从 bin 目录执行下面的命令安装一个 GUI。请确保系统连接到互联网：

```
plugin -install mobz/elasticsearch-head
```

6. 现在，可以使用 URLhttp://localhost:9200/_plugin/head/来访问 GUI 界面。

可以把 localhost 和 **9200** 替换为你各自的主机名和端口号。

安装 Logstash

我们可以按给定的步骤安装 Logstash：

1. 从 https://www.elastic.co/downloads/logstash 下载最新的 Logstash 软件。

2. 将其解压缩到你的系统中的所需位置。

   编写一个配置文件，如下所示。它指示 Logstash 从给定文件读取输入并将其传递给 Elasticsearch（见以下 config 文件；Elasticsearch 是用 localhost 和 9200 端口表示的）。它是最简单的配置文件。要添加过滤器，并了解更多关于 Logstash 的信息，你可以查看 Logstash 参考文档，它位于 https://www.elastic.co/guide/en/logstash/current/ index.html。

正如你可以看到，OTRS service 日志和 edge-server 日志被作为输入添加。同样，你还可以添加其他的微服务日志文件。

```
input {
  ### OTRS ###
  file {
    path => "\logs\otrs-service.log"
    type => "otrs-api"
    codec => "json"
    start_position => "beginning"
  }

  ### 边缘 ###
  file {
    path => "/logs/edge-server.log"
    type => "edge-server"
    codec => "json"
  }
}

output {
```

```
  stdout {
    codec => rubydebug
  }
  elasticsearch {
    hosts => "localhost:9200"
  }
}
```

3. 转到 Logstash 主目录，在基于 Unix 的系统上，运行 `bin/logstash agent -f logstash.conf`，而在 Windows 上，运行 `bin/logstash.bat agent -f logstash. conf`。在这里，Logstash 使用 `agent` 命令执行。Logstash 代理从配置文件输入字段中提供的来源收集数据，并将其输出发送到 Elasticsearch。在这里，我们不使用过滤器，否则它可能会在把输入的数据提供给 Elasticsearch 之前对它做处理。

安装 Kibana

我们可以通过下面给定的步骤安装 Kibana web 应用程序：

1. 从 `https://www.elastic.co/ downloads/kibana` 下载最新 Kibana 软件。
2. 将其解压缩到你的系统中的所需位置。
3. 从 Kibana 主目录打开配置文件 `config/kibana.yml`，并把 `elasticsearch.url` 指向以前配置的 Elasticsearch 实例：

   ```
   elasticsearch.url:"http://localhost:9200"
   ```

4. 转到 Kibana 主目录，在基于 Unix 的系统上，运行 `bin/kibana agent -f logstash.conf`；而在 Windows 上，运行 `bin/kibana.bat agent -f logstash.conf`。
5. 现在可以从你的浏览器使用 URL `http://localhost:5601/`访问 Kibana 应用程序。

   为了解 Kibana 的更多内容，查看在 `https://www.elastic.co/guide/en/kibana/current/getting-started. html` 的 Kibana 参考文档。

在我们按照上述步骤操作时，你可能会注意到，这需要一定的工作量。如果你想要避免手动安装，可以 Docker 化它。如果你不想要投入精力创建 ELK 环境的 Docker 容器，可以从 Docker Hub 选择一个。在 Docker Hub 有许多现成的 ELK 环境的 Docker 映像，可以尝试不同的ELK容器并选择最适合的。willdurand/elk是被下载得最多的容器，也容易启动，并良好地与 Docker Compose 配合工作。

有关 ELK 环境实现的提示

- 为了避免任何的数据丢失和处理突然激增的输入负载，建议在 Logstash 和 Elasticsearch 之间使用代理，如 Redis 或 RabbitMQ。
- 如果你正在使用集群，防止脑裂问题，请使用奇数个 Elasticsearch 的节点数。
- 在 Elasticsearch 中，对于给定的数据始终使用合适的字段类型。这将允许你执行不同的检查，例如，`int` 字段类型将允许你执行（`"http_status: < 400"`）或（`"http_status:=200"`）检查。同样，其他字段类型还允许你执行类似的检查。

服务调用关联 ID 的使用

当你对任何 REST 端点发出调用并有任何问题弹出时，很难跟踪问题和它的起源，因为向服务器发出每个调用时，此调用可能调用另一个，以此类推。这使得很难弄清楚一个特定的请求如何被变换，以及它调用了什么。通常情况下，由一个服务造成的问题可能会造成其他位置的服务出问题。这个问题很难跟踪，可能需要大量的工作。如果它是整体式的，你会知道正确的调查方向，但微服务使得难以理解问题的根源是什么和你应该在哪里得到你的数据。

让我们看看怎样解决这个问题

利用跨所有调用传递的关联 ID，能够跟踪每个请求，并轻松地跟踪路由，每个请求都具有其独特的关联 ID。因此，当我们调试任何问题时，关联 ID 都是我们的出发点。我们可以跟着它一路跟踪，就可以找出到底什么东西出错了。关联 ID 需要一些额外的开发工作，但这个工作非常值得做，从长远来看将有很大帮助。当一个请求在不同的微服务之间传输时，你将能够看到所有交互和哪个服务有问题。

这不是新东西或微服务的发明。许多受欢迎的产品，如 Microsoft SharePoint 都已经采用了这种模式。

依赖项和版本

在产品开发中，我们面临的两个常见的问题是循环依赖和 API 版本。我们将在微服务的基础架构中讨论它们。

循环依赖关系及其影响

一般来说，整体式架构具有典型的分层模型，而微服务带有图状模型。因此，微服务可能会有循环依赖。

因此，有必要对微服务的关系保持一个依赖项检查。

让我们看看以下两种情况：

- 如果你的微服务之间有一个循环依赖，当某一具体事务可能会被困在一个循环中时，你很容易遇到分布式的堆栈溢出错误。例如，当餐馆的餐桌由一个人预订时。在这种情况下，餐馆需要知道预订的人（`findBookedUser`），而在给定的时间，人也需要了解餐馆（`findBookedRestaurant`）。如果这些服务的设计不好，就可能在循环中互相调用。结果可能是由 JVM 生成的堆栈溢出。
- 如果两个服务共享一个依赖项，而你更新另一个服务的 API 的方式可能会影响它们，你会需要一次更新所有三个服务。这就引出了一些问题，如你应该首先更新哪个服务？此外，你怎么使这个成为一个安全事务？

设计系统时需要分析它

因此，在设计微服务时，需要在不同的服务间建立正确的关系，以避免任何循环依赖是非常重要的。它是一个设计问题，并且必须得到解决，即使这需要重构代码。

维护不同版本

当你有更多的服务时，这意味着每个服务有不同的发布周期，这将通过不同版本服务

的引入增加复杂性，相同的 REST 服务将有不同的版本。当某个问题在一个版本中已经消失并在一个新版本中复现的时候，再现问题的解决办法将是很难的。

让我们了解更多

API 版本控制之所以重要是因为随着时间的推移 API 会改变。随着时间的推移，你的知识和经验增加，并导致 API 的改变。改变的 API 可能会破坏现有客户端集成。

因此存在管理 API 版本的各种方法。其中一种方法是在路径中使用版本，就像我们在本书中已经使用的那样，还有一些则使用 HTTP 标头。HTTP 标头可以是自定义请求标头，或者你可以使用 Accept Header 表示调用 API 的版本。更多关于如何使用 HTTP 标头来处理版本的信息，请参阅 Packt 出版的 Bhakti Mehta 编写的《*RESTful Java Patterns and Best Practices*》一书：`https://www.packtpub.com/application-development/restful-java-patterns-and-best-practices`。

在实现你的微服务时，在对任何问题进行故障排除时，在日志中产生版本号是非常重要的。此外，理想情况下，应该避免你的任何微服务的任何实例有太多版本。

参考资料

如下链接将有更多的信息：

- Elasticsearch：`https://www.elastic.co/products/elasticsearch`
- Logstash：`https://www.elastic.co/products/logstash`
- Kibana：`https://www.elastic.co/products/kibana`
- willdurand/elk：ELK Docker 映像
- *Mastering Elasticsearch-Second Edition*：`https://www.packtpub.com/webdevelopment/mastering-elasticsearch-second-edition`

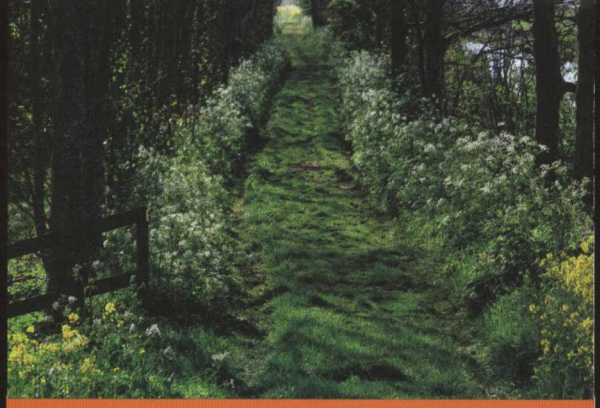
小结

在本章中，我们探讨了 ELK 环境的概念和安装。在 ELK 环境中，Elasticsearch 用来存放日志和来自 Kibana 的服务查询。Logstash 是一个希望从中收集来自每个服务器的运行日

志的代理。Logstash 可以读取日志、过滤/转换它们，并提供给 Elasticsearch。Kibana 读取/查询来自 Elasticsearch 的数据并以表格或图形的可视化效果呈现。

我们也了解了调试问题时拥有关联 ID 的实用性。在这一章的结束，我们也探讨了几种微服务设计的缺点。在本书中包括有关微服务的所有话题是具有挑战性的，所以我试图尽可能在相关章节中包含参考资料，使你可以探索更多的相关信息。现在我想让你开始在工作场所或在个人项目中实现我们在这一章已经学到的概念。这不但会给你亲身的体验，而且也可以让你掌握微服务。此外，学会了这些内容，你就会有能力参与当地的聚会和会议。

Broadview
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版



Java微服务

随着云平台的采用，企业应用程序的开发从整体应用程序转移到小型、轻量和过程驱动的组件，这种组件称为微服务。微服务是设计可扩展、易于维护的应用程序的下一个重大事件。它们不但使应用程序开发起来更容易，而且还提供了极大的灵活性来以最佳方式利用各种资源。

本书帮助你构建供企业使用的微服务架构实现。从核心概念和框架开始介绍，然后着重讲述大型软件项目的高层次设计，逐渐进入开发环境的设置和前期配置，对微服务架构进行持续集成的部署。然后使用Spring Security实现微服务的安全性，利用REST Java客户端和其他工具有效地执行测试。最后，展示了微服务设计的最佳做法和一般原则，以及如何检测和调试开发过程出现的问题。

本书的受众

如果您是一位熟悉微服务架构的Java开发人员，并对微服务的核心要素和应用程序有合理的知识水平和理解，现在想要深入了解如何有效地实施企业级微服务，那么本书适合您。

本书内容提要

- 使用领域驱动设计方法来设计和实现微服务
- 使用Spring Security实现微服务的安全性
- 部署和测试微服务
- 检测和调试开发过程出现的问题
- 利用JavaScript的Web应用程序来使用微服务
- 学习关于微服务的最佳做法和一般原则

上架建议：编程语言>Java

[PACKT] open source*
PUBLISHING
community experience distilled

ISBN 978-7-121-30493-4
9 787121 304934 >
定价：69.00元

PHEI
责任编辑：张春雨
封面设计：李　玲